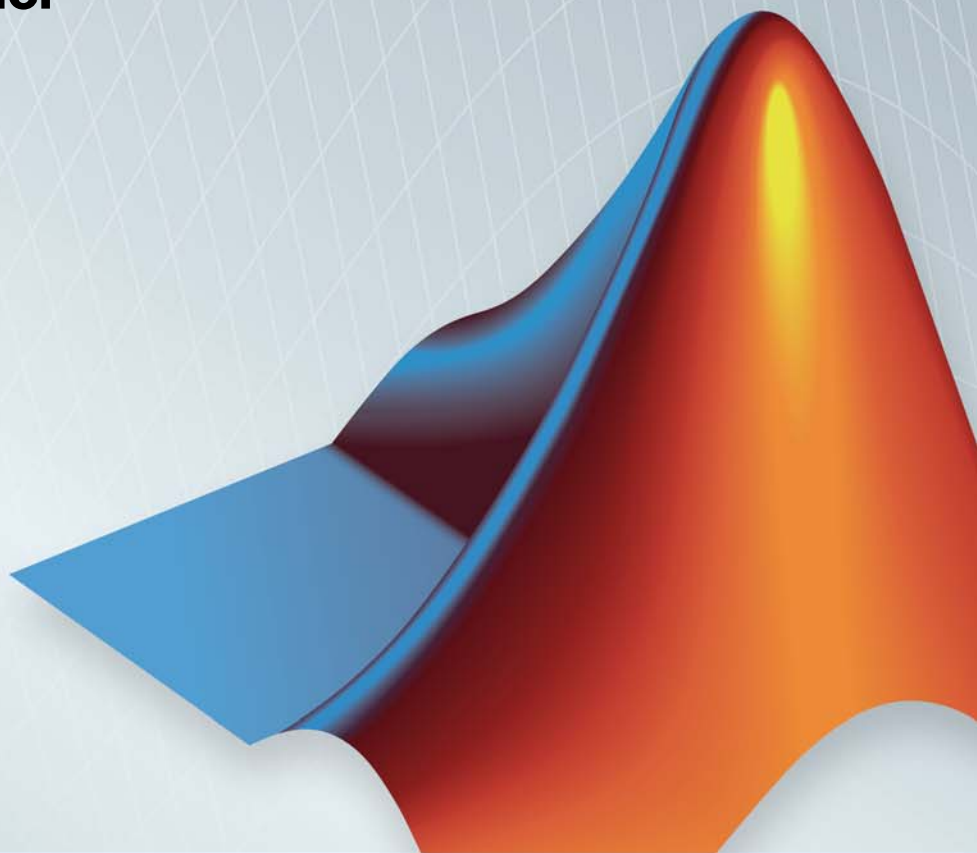


MATLAB® Coder™

User's Guide

R2013a



MATLAB®



How to Contact MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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.

MATLAB® Coder™ User's Guide

© COPYRIGHT 2011–2013 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 2 (R2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.4 (Release 2013a)

Check Bug Reports for Issues and Fixes

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 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.

About MATLAB Coder

1

Product Description	1-2
Key Features	1-2
Product Overview	1-3
When to Use MATLAB Coder	1-3
Code Generation for Embedded Software Applications ...	1-3
Code Generation for Fixed-Point Algorithms	1-4
Code Generation Workflow	1-5
See Also	1-5

Design Considerations for C/C++ Code Generation

2

When to Generate Code from MATLAB Algorithms ...	2-2
When Not to Generate Code from MATLAB Algorithms ..	2-2
Which Code Generation Feature to Use	2-4
Prerequisites for C/C++ Code Generation from MATLAB	2-5
MATLAB Code Design Considerations for Code Generation	2-6
See Also	2-7
Expected Differences in Behavior After Compiling MATLAB Code	2-8

Why Are There Differences?	2-8
Character Size	2-8
Order of Evaluation in Expressions	2-8
Termination Behavior	2-9
Size of Variable-Size N-D Arrays	2-9
Size of Empty Arrays	2-10
Floating-Point Numerical Results	2-10
NaN and Infinity Patterns	2-11
Code Generation Target	2-11
MATLAB Class Initial Values	2-11
Variable-Size Support for Code Generation	2-11

MATLAB Language Features Supported for C/C++ Code Generation	2-12
MATLAB Language Features Not Supported for C/C++ Code Generation	2-13

System Objects Supported for Code Generation

3

System Objects Supported for Code Generation	3-2
Code Generation for System Objects	3-2
Computer Vision System Toolbox System Objects	3-2
Communications System Toolbox System Objects	3-7
DSP System Toolbox System Objects	3-13

Functions Supported for Code Generation

4

Functions Supported for Code Generation — Alphabetical List	4-2
--	-----

Functions Supported for Code Generation — Categorical List	4-79
Aerospace Toolbox Functions	4-80
Arithmetic Operator Functions	4-81

Bit-Wise Operation Functions	4-81
Casting Functions	4-82
Communications System Toolbox Functions	4-82
Complex Number Functions	4-82
Computer Vision System Toolbox Functions	4-83
Data and File Management Functions	4-84
Data Type Functions	4-85
Derivative and Integral Functions	4-85
Discrete Math Functions	4-85
Error Handling Functions	4-86
Exponential Functions	4-86
Filtering and Convolution Functions	4-87
Fixed-Point Designer Functions	4-87
Histogram Functions	4-96
Image Processing Toolbox Functions	4-96
Input and Output Functions	4-98
Interpolation and Computational Geometry Functions ...	4-99
Linear Algebra	4-99
Logical Operator Functions	4-99
MATLAB Compiler Functions	4-100
MATLAB Desktop Environment Functions	4-100
Matrix and Array Functions	4-100
Nonlinear Numerical Methods	4-104
Polynomial Functions	4-105
Relational Operator Functions	4-105
Rounding and Remainder Functions	4-105
Set Functions	4-106
Signal Processing Functions in MATLAB	4-106
Signal Processing Toolbox Functions	4-107
Special Values	4-111
Specialized Math	4-112
Statistical Functions	4-112
String Functions	4-113
Structure Functions	4-114
Trigonometric Functions	4-114

Defining MATLAB Variables for C/C++ Code Generation

5

Variables Definition for Code Generation	5-2
--	-----

Best Practices for Defining Variables for C/C++ Code	
Generation	5-3
Define Variables By Assignment Before Using Them	5-3
Use Caution When Reassigning Variables	5-6
Use Type Cast Operators in Variable Definitions	5-6
Define Matrices Before Assigning Indexed Variables	5-6
Eliminate Redundant Copies of Variables in Generated Code	5-7
When Redundant Copies Occur	5-7
How to Eliminate Redundant Copies by Defining	
Uninitialized Variables	5-7
Defining Uninitialized Variables	5-8
Reassignment of Variable Properties	5-9
Define and Initialize Persistent Variables	5-10
Reuse the Same Variable with Different Properties ...	5-11
When You Can Reuse the Same Variable with Different	
Properties	5-11
When You Cannot Reuse Variables	5-12
Limitations of Variable Reuse	5-14
Avoid Overflows in for-Loops	5-16
Supported Variable Types	5-18

Defining Data for Code Generation

6

Data Definition for Code Generation	6-2
Code Generation for Complex Data	6-4
Restrictions When Defining Complex Variables	6-4
Expressions Containing Complex Operands Yield Complex	
Results	6-5

Code Generation for Characters	6-6
---	------------

Code Generation for Variable-Size Data

7

What Is Variable-Size Data?	7-2
Variable-Size Data Definition for Code Generation ...	7-3
Bounded Versus Unbounded Variable-Size Data	7-4
Control Memory Allocation of Variable-Size Data	7-5
Specify Variable-Size Data Without Dynamic Memory Allocation	7-6
Fixing Upper Bounds Errors	7-6
Specifying Upper Bounds for Variable-Size Data	7-6
Variable-Size Data in Code Generation Reports	7-10
What Reports Tell You About Size	7-10
How Size Appears in Code Generation Reports	7-11
How to Generate a Code Generation Report	7-11
Define Variable-Size Data for Code Generation	7-12
When to Define Variable-Size Data Explicitly	7-12
Using a Matrix Constructor with Nonconstant Dimensions	7-13
Inferring Variable Size from Multiple Assignments	7-13
Defining Variable-Size Data Explicitly Using <code>coder.varsize</code>	7-14
C Code Interface for Arrays	7-19
C Code Interface for Statically Allocated Arrays	7-19
C Code Interface for Dynamically Allocated Arrays	7-20
Utility Functions for Creating <code>emxArray</code> Data Structures	7-21

Diagnose and Fix Variable-Size Data Errors	7-23
Diagnosing and Fixing Size Mismatch Errors	7-23
Diagnosing and Fixing Errors in Detecting Upper Bounds	7-25
Incompatibilities with MATLAB in Variable-Size	
Support for Code Generation	7-27
Incompatibility with MATLAB for Scalar Expansion	7-27
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays	7-29
Incompatibility with MATLAB in Determining Size of Empty Arrays	7-30
Incompatibility with MATLAB in Vector-Vector Indexing	7-31
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation	7-32
Incompatibility with MATLAB in Concatenating Variable-Size Matrices	7-33
Dynamic Memory Allocation Not Supported for MATLAB Function Blocks	7-33
Restrictions on Variable Sizing in Toolbox Functions	
Supported for Code Generation	7-34
Common Restrictions	7-34
Toolbox Functions with Variable Sizing Restrictions	7-35

Code Generation for MATLAB Structures

8

Structure Definition for Code Generation	8-2
Structure Operations Allowed for Code Generation ...	8-3
Define Scalar Structures for Code Generation	8-4
Restriction When Using <code>struct</code>	8-4
Restrictions When Defining Scalar Structures by Assignment	8-4
Adding Fields in Consistent Order on Each Control Flow Path	8-4

Restriction on Adding New Fields After First Use	8-5
Define Arrays of Structures for Code Generation	8-7
Ensuring Consistency of Fields	8-7
Using repmat to Define an Array of Structures with Consistent Field Properties	8-7
Defining an Array of Structures Using Concatenation	8-8
Make Structures Persistent	8-9
Index Substructures and Fields	8-10
Assign Values to Structures and Fields	8-12
Pass Large Structures as Input Parameters	8-13

Code Generation for Enumerated Data

9

Enumerated Data Definition for Code Generation	9-2
Enumerated Types Supported for Code Generation ...	9-3
Enumerated Type Based on int32	9-3
When to Use Enumerated Data for Code Generation ..	9-5
Generate Code for Enumerated Data from MATLAB	
Algorithms	9-6
How to Generate Code for Enumerated Data	9-6
Define Enumerated Data for Code Generation	9-8
Naming Enumerated Types for Code Generation	9-9
Instantiate Enumerated Types for Code Generation ..	9-10

Operations on Enumerated Data Allowed for Code	
Generation	9-11
Assignment Operator, =	9-11
Relational Operators, < > <= >= == ~=	9-11
Cast Operation	9-12
Indexing Operation	9-12
Control Flow Statements: if, switch, while	9-13
Include Enumerated Data in Control Flow	
Statements	9-14
if Statement with Enumerated Data Types	9-14
switch Statement with Enumerated Data Types	9-15
while Statement with Enumerated Data Types	9-18
Customize Enumerated Types Based on int32	9-20
About Customizing Enumerated Types	9-20
Specify a Default Enumerated Value	9-22
Specify a Header File	9-23
Control Names of Enumerated Type Values in	
Generated Code	9-26
Change and Reload Enumerated Data Types	9-28
Restrictions on Use of Enumerated Data in	
for-Loops	9-29
Toolbox Functions That Support Enumerated Types for	
Code Generation	9-30

Code Generation for MATLAB Classes

10

MATLAB Classes Definition for Code Generation	10-2
Language Limitations	10-2
Code Generation Features Not Compatible with Classes ..	10-3
Defining Class Properties for Code Generation	10-4
Calls to Base Class Constructor	10-6

Classes That Support Code Generation	10-8
Generate Code for MATLAB Value Classes	10-9
Generate Code for MATLAB Handle Classes and System Objects	10-15
MATLAB Classes in Code Generation Reports	10-17
What Reports Tell You About Classes	10-17
How Classes Appear in Code Generation Reports	10-17
How to Generate a Code Generation Report	10-19
Troubleshooting Issues with MATLAB Classes	10-20
Class <i>class</i> does not have a property with name <i>name</i> ...	10-20

Code Generation for Function Handles

11

Function Handle Definition for Code Generation	11-2
Define and Pass Function Handles for Code Generation	11-3
Function Handle Limitations for Code Generation ...	11-5

Defining Functions for Code Generation

12

Specify Variable Numbers of Arguments	12-2
Supported Index Expressions	12-3

Apply Operations to a Variable Number of Arguments	12-4
When to Force Loop Unrolling	12-4
Using Variable Numbers of Arguments in a for-Loop	12-5
Implement Wrapper Functions	12-7
Passing Variable Numbers of Arguments from One Function to Another	12-7
Pass Property/Value Pairs	12-8
Variable Length Argument Lists for Code Generation	12-10

Calling Functions for Code Generation

13

Resolution of Function Calls in MATLAB Generated Code	13-2
Key Points About Resolving Function Calls	13-4
Compile Path Search Order	13-4
When to Use the Code Generation Path	13-5
Resolution of File Types on Code Generation Path ...	13-6
Compilation Directive %#codegen	13-8
Call Local Functions	13-9
Call Supported Toolbox Functions	13-10
Call MATLAB Functions	13-11
Declaring MATLAB Functions as Extrinsic Functions ...	13-12
Calling MATLAB Functions Using feval	13-16
How MATLAB Resolves Extrinsic Functions During Simulation	13-16
Working with mxArray	13-17

Restrictions on Extrinsic Functions for Code Generation ..	13-19
Limit on Function Arguments	13-19

Fixed-Point Conversion

14

Convert MATLAB Code to Fixed-Point C Code	14-2
Propose Fixed-Point Data Types Based on Simulation	
Ranges	14-3
Propose Fixed-Point Data Types Based on Derived	
Ranges	14-20
Specify Type Proposal Options	14-34
Log Histogram Data	14-37
View and Modify Variable Information	14-40
View Variable Information	14-40
Modify Variable Information	14-40
Revert Changes	14-42
Promote Sim Min and Sim Max Values	14-43
Build Instrumented MEX Function	14-44
Propose Fixed-Point Data Types	14-45
Apply Fixed-Point Data Types	14-54
Modify Data Type Proposal Settings	14-60
Modify Instrumentation Report Settings	14-64
Automated Fixed-Point Conversion	14-65

License Requirements	14-65
Fixed-Point Conversion Capabilities	14-65
Proposing Data Types	14-67
Viewing Functions	14-68
Viewing Variables	14-69
Histogram	14-70
Function Replacements	14-71
Validating Types	14-72
Testing Numerics	14-72
Instrumented MEX Functions	14-73
Generating Instrumented MEX Functions	14-73
Merging Instrumentation Results	14-73
Clearing Instrumentation Results	14-74
Redirecting Entry-Point Calls to MEX Function	14-74
Proposing Fraction Lengths	14-74
Proposing Word Lengths	14-74

Bug Reports

15

Check Bug Reports for Issues and Fixes	15-2
--	------

Setting Up a MATLAB Coder Project

16

MATLAB Coder Project Set Up Workflow	16-2
Creating a New Project	16-3
From the MATLAB APPS Tab	16-3
At the Command Line	16-3
From a MATLAB Coder Project	16-4
Opening an Existing Project	16-5
From the MATLAB APPS Tab	16-5

At the Command Line	16-5
From a MATLAB Coder Project	16-5
Adding Files to the Project	16-6
Specifying Properties of Primary Function Inputs in a Project	16-7
Why You Must Specify Input Properties	16-7
How to Specify an Input Definition in a Project	16-7
Autodefine Input Types	16-8
How MATLAB Coder Autodefines Input Types	16-8
Prerequisites for Autodefining Input Types	16-8
How to Autodefine Input Types	16-8
Define Input Parameters by Example in a Project	16-12
How to Define an Input Parameter by Example	16-12
Specifying Input Parameters by Example	16-13
Specifying an Enumerated Type Input Parameter by Example	16-15
Specifying a Fixed-Point Input Parameter by Example ...	16-17
Define or Edit Input Parameter Type in a Project	16-19
How to Define or Edit an Input Parameter Type	16-19
Specifying an Enumerated Type Input Parameter by Type	16-21
Specifying a Fixed-Point Input Parameter by Type	16-21
Specifying Structures	16-23
Define Constant Input Parameters in a Project	16-30
Define Inputs Programmatically in the MATLAB File	16-31
Adding Global Variables in a Project	16-32
Specifying Global Variable Type and Initial Value in a Project	16-33
Why Specify a Type Definition for Global Variables?	16-33
How to Specify a Global Variable Type	16-33

Defining a Global Variable by Example	16-34
Defining or Editing Global Variable Type	16-35
Defining Global Variable Initial Value	16-37
Removing Global Variables	16-39
Specify Output File Name	16-40
Command Line Alternative	16-40
Specify Output File Locations	16-41
Command Line Alternative	16-41
Selecting Output Type	16-42
Command Line Alternative	16-42
Changing Output Type	16-42

Preparing MATLAB Code for C/C++ Code Generation

17

Workflow for Preparing MATLAB Code for Code Generation	17-2
See Also	17-3
Fixing Errors Detected at Design Time	17-4
See Also	17-4
Using the Code Analyzer	17-5
Check Code With the Code Analyzer	17-6
Check Code Using the Code Generation Readiness Tool	17-8
Run Code Generation Readiness Tool at the Command Line	17-8
Run Code Generation Readiness Tool from the Current Folder Browser	17-8
Run the Code Generation Readiness Tool in a Project	17-9

See Also	17-9
Code Generation Readiness Tool	17-10
What Information Does the Code Generation Readiness Tool Provide?	17-10
Summary Tab	17-11
Code Structure Tab	17-12
See Also	17-15
Unable to Determine Code Generation Readiness	17-16
Generate MEX Functions Using the MATLAB Coder	
Project Interface	17-17
Project Workflow for Generating MEX Functions	17-17
Generate MEX Functions Using the Project Interface	17-17
Configure Project Settings	17-22
Build a MATLAB Coder Project	17-23
See Also	17-24
Generate MEX Functions at the Command Line	17-25
Command-line Workflow for Generating MEX Functions	17-25
Generate MEX Functions at the Command Line	17-25
Generating MEX Functions at the Command Line Using codegen	17-26
See Also	17-26
Fix Errors Detected at Code Generation Time	17-27
See Also	17-27
Design Considerations When Writing MATLAB Code for Code Generation	17-28
See Also	17-29
Running MEX Functions	17-30
Debugging MEX Functions	17-30
Debugging Strategies	17-31

Workflow for Testing MEX Functions in MATLAB	18-2
See Also	18-2
Why Test MEX Functions in MATLAB?	18-4
Running MEX Functions	18-5
Debugging MEX Functions	18-5
Verify MEX Functions in a Project	18-6
Using Test Files That Call Only MATLAB Functions	18-6
Using Test Files That Call MEX Functions	18-7
Verify MEX Functions at the Command Line	18-8
Debug Run-Time Errors	18-9
Viewing Errors in the Run-Time Stack	18-9
Handling Run-Time Errors	18-11

Generating C/C++ Code from MATLAB Code

Code Generation Workflow	19-3
See Also	19-4
C/C++ Code Generation	19-5
Specify Custom Files to Build	19-5
Generating C/C++ Static Libraries from MATLAB	
Code	19-7
Generate a C Static Library Using the Project Interface . .	19-7
Generate a C Static Library at the Command Line	19-10

Generating C/C++ Dynamically Linked Libraries from	
MATLAB Code	19-11
Dynamic Libraries Generated by MATLAB Coder	19-11
Generate a C Dynamically Linked Library (DLL) Using the Project Interface	19-11
Generate a C Dynamic Library at the Command Line	19-13
Generating Standalone C/C++ Executables from	
MATLAB Code	19-15
Generate a C Executable Using the Project Interface	19-15
Generate a C Executable at the Command Line	19-17
Specifying main Functions for C/C++ Executables	19-19
Specify main Functions	19-19
Build Setting Configuration	19-21
Specify Output Type	19-21
Specify a Language for Code Generation	19-24
Specify Output File Name	19-25
Specify Output File Locations	19-26
Parameter Specification Methods	19-27
Specify Build Configuration Parameters	19-28
Share Build Configuration Settings	19-35
Export Settings	19-35
Import Settings	19-36
See Also	19-37
Primary Function Input Specification	19-38
Why You Must Specify Input Properties	19-38
Properties to Specify	19-38
Rules for Specifying Properties of Primary Inputs	19-42
Methods for Defining Properties of Primary Inputs	19-42
Define Input Properties by Example at the Command Line	19-43
Specify Constant Inputs at the Command Line	19-46
Specify Variable-Size Inputs at the Command Line	19-48
Define Input Properties Programmatically in the	
MATLAB File	19-50
How to Use assert with MATLAB Coder	19-50
Rules for Using assert Function	19-57
Specifying General Properties of Primary Inputs	19-58

Specifying Properties of Primary Fixed-Point Inputs	19-59
Specifying Class and Size of Scalar Structure	19-59
Specifying Class and Size of Structure Array	19-60
Speed Up Compilation	19-61
Generate Code Only	19-61
Disable Compiler Optimization	19-61
Code Optimization	19-63
Unroll for-loops	19-63
Inline Code	19-65
Eliminate Redundant Copies of Function Inputs (A=foo(A))	19-66
Rewrite Logical Array Indexing as a Loop	19-68
Paths and File Infrastructure Setup	19-69
Compile Path Search Order	19-69
Specifying Folders to Search for Custom Code	19-69
Naming Conventions	19-70
Generate Code for Multiple Entry-Point Functions ...	19-75
Advantages of Generating Code for More Than One Entry-Point Function	19-75
Generating Code for More Than One Entry-Point Function Using the Project Interface	19-75
Generating Code for More Than One Entry-Point Function at the Command Line	19-78
How to Call an Entry-Point Function in a MEX Function ..	19-79
How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code	19-80
Generate Code for Global Data	19-81
Workflow	19-81
Declare Global Variables	19-81
Define Global Data	19-82
Synchronizing Global Data with MATLAB	19-83
Limitations of Using Global Data	19-87
Generation of Traceable Code	19-88
About Code Traceability	19-88
Generate Traceable Code	19-89

Format of Traceability Tags	19-91
Location of Comments in Generated Code	19-91
Traceability Limitations	19-96
Generate Code for Enumerated Types	19-97
Generate Code for Variable-Size Data	19-98
Disable Support for Variable-Size Data	19-98
Control Dynamic Memory Allocation	19-99
Generating Code for MATLAB Functions with Variable-Size Data	19-101
Generate Code for a MATLAB Function That Expands a Vector in a Loop	19-103
Using Dynamic Memory Allocation for an "Atoms" Simulation	19-110
Code Generation for MATLAB Classes	19-118
How MATLAB Coder Partitions Generated Code	19-119
Partitioning Generated Files	19-119
How to Select the File Partitioning Method	19-119
Partitioning Generated Files with One C/C++ File Per MATLAB File	19-120
Generated Files and Locations	19-125
File Partitioning and Inlining	19-128
Customize the Post-Code-Generation Build Process ..	19-133
Workflow for Customizing Post-Code-Generation Builds ..	19-133
Build Information Object	19-133
Build Information Functions	19-134
Programming a Post-Code-Generation Command	19-172
Using a Post-Code-Generation Command in Your Build ..	19-172
Programming and Using a Post-Code-Generation Command at the Command Line	19-174
Code Generation Reports	19-176
About Code Generation Reports	19-176
Enable Code Generation Reports	19-179
View Your MATLAB Code in a Report	19-180
Viewing Call Stack Information	19-181
View Generated C/C++ Code in a Report	19-184

Viewing the Build Summary Information	19-184
View Error and Warning Messages in a Report	19-185
Viewing Variables in Your MATLAB Code	19-186
Viewing Target Build Information	19-192
Keyboard Shortcuts for the Code Generation Report	19-193
Report Limitations	19-193
Troubleshooting	19-195
Run-time Stack Overflow	19-195
Package Code For Use in Another Development	
Environment	19-196
When to Package Code	19-196
Package Generated Code in a Project	19-196
Package Generated Code at the Command Line	19-198

Custom Toolchain Registration

20

Adding a Custom Toolchain	20-2
Introduction	20-2
Create a toolchain specification file	20-2
Create a handle object for ToolchainInfo	20-2
Populate ToolchainInfo	20-4
Populate ToolchainInfo's Build Tools	20-4
Populate ToolchainInfo's Build Configurations	20-8
Register a Toolchain	20-9
Create a ToolchainInfo MAT file	20-9
Define toolchain in rtwTargetInfo.m	20-9
Provide info about the toolchain	20-10
Reset the toolchain registry	20-10
Determine if a Toolchain is Registered	20-11
Command Line Approach	20-11
Using a Custom Toolchain — Command Line	
Approach	20-12

Using a Custom Toolchain — UI Approach	20-13
Removing a Toolchain	20-14
Troubleshooting	20-15
About ToolchainInfo	20-16
What is a Toolchain?	20-16
What is ToolchainInfo?	20-16

Deploying Generated Code

21

Call a C Static Library Function from C Code	21-2
Call a C/C++ Static Library Function from MATLAB Code	21-4
Call Generated C/C++ Functions	21-6
Conventions for Calling Functions in Generated Code	21-6
How to Call C/C++ Functions from MATLAB Code	21-6
Calling Initialize and Terminate Functions	21-7
Calling C/C++ Functions with Multiple Outputs	21-8
Calling C/C++ Functions that Return Arrays	21-8
Use a MATLAB Coder Dynamic Library in a Simple Microsoft Visual Studio Project	21-9
Custom C/C++ Code Integration	21-12
About Custom C/C++ Code Integration with MATLAB Coder	21-12
Specifying Custom C/C++ Files in the Project Settings Dialog Box	21-12
Specifying Custom C/C++ Files at the Command Line	21-13
Specifying Custom C/C++ Files with Configuration Objects	21-13

Workflow for Accelerating MATLAB Algorithms	22-2
See Also	22-3
Edge Detection on Images	22-4
Accelerate MATLAB Algorithms	22-11
Modifying MATLAB Code for Acceleration	22-12
How to Modify Your MATLAB Code for Acceleration	22-12
Unroll for-loops	22-12
Inline Code	22-14
Eliminate Redundant Copies of Function Inputs (A=foo(A))	22-15
Control Run-Time Checks	22-18
Types of Run-Time Checks	22-18
When to Disable Run-Time Checks	22-19
How to Disable Run-Time Checks	22-19
Acceleration of MATLAB Algorithms Using Parallel	
for-loops (parfor)	22-21
Parallel for-loops (parfor) in MEX Functions	22-21
When to Use parfor-loops	22-22
When Not to Use parfor-loops	22-23
Control Compilation of parfor-loops	22-23
Supported Compilers	22-24
parfor-Loop Syntax and Restrictions	22-24
parfor Limitations	22-25
Reduction Assignments in parfor-loops	22-28
What are Reduction Assignments?	22-28
Multiple Reductions in a parfor-loop	22-28
Classification of Variables in parfor-loops	22-29
Overview	22-29
Sliced Variables	22-30
Broadcast Variables	22-32

Reduction Variables	22-32
Temporary Variables	22-38
Accelerate MATLAB Algorithms That Use Parallel for-loops (parfor)	22-40
Accelerate MATLAB Algorithms That Use Parallel for-loops (parfor) Specifying the Maximum Number of Threads	22-41
Troubleshooting parfor-loops	22-42
What Causes Errors About the Use of Global Structures in Parallel Regions?	22-42
Compiler Does Not Support OpenMP	22-42
Accelerating Simulation of Bouncing Balls	22-44

Calling C/C++ Functions from Generated Code

23

MATLAB Coder Interface to C/C++ Code	23-2
How to Call C/C++ Code from Generated Code	23-2
Why Call C/C++ Functions from Generated Code?	23-2
Call External C/C++ Functions	23-3
Pass Arguments by Reference to External C/C++ Functions	23-3
Manipulate C Data	23-5
Call External C/C++ Functions	23-7
Workflow for Calling External C/C++ Functions	23-7
Best Practices for Calling C/C++ Code from Generated Code	23-8
Return Multiple Values from C Functions	23-9
How MATLAB Coder Infers C/C++ Data Types	23-10
Mapping MATLAB Types to C/C++	23-10

Mapping embedded.numerictypes to C/C++	23-11
Mapping Arrays to C/C++	23-12
Mapping Complex Values to C/C++	23-12
Mapping Structures to C/C++ Structures	23-13
Mapping Strings to C/C++	23-14
Mapping Multiword Types to C/C++	23-14

Index

About MATLAB Coder

- “Product Description” on page 1-2
- “Product Overview” on page 1-3
- “Code Generation Workflow” on page 1-5

Product Description

Generate C and C++ code from MATLAB® code

MATLAB Coder™ generates standalone C and C++ code from MATLAB code. The generated source code is portable and readable. MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. It can generate MEX functions that let you accelerate computationally intensive portions of MATLAB code and verify the behavior of the generated code.

Key Features

- ANSI®/ISO® compliant C and C++ code generation
- MEX function generation for fixed-point and floating-point math
- Project management tool for specifying entry points, input data properties, and other code-generation configuration options
- Static or dynamic memory allocation for variable-size data
- Code generation support for many functions and System objects in Communications System Toolbox™, DSP System Toolbox™, and Computer Vision System Toolbox™
- Support for common MATLAB language features, including matrix operations, subscripting, program controls statements (if, switch, for, while), and structures

Product Overview

In this section...
“When to Use MATLAB® Coder™” on page 1-3
“Code Generation for Embedded Software Applications” on page 1-3
“Code Generation for Fixed-Point Algorithms” on page 1-4

When to Use MATLAB Coder

Use MATLAB Coder to:

- Generate readable, efficient, standalone C/C++ code from MATLAB code.
- Generate MEX functions from MATLAB code to:
 - Accelerate your MATLAB algorithms.
 - Verify generated C code within MATLAB.
- Integrate custom C/C++ code into MATLAB.

Code Generation for Embedded Software Applications

The Embedded Coder® product extends the MATLAB Coder product with features that are important for embedded software development. Using the Embedded Coder add-on product, you can generate code that has the clarity and efficiency of professional handwritten code. For example, you can:

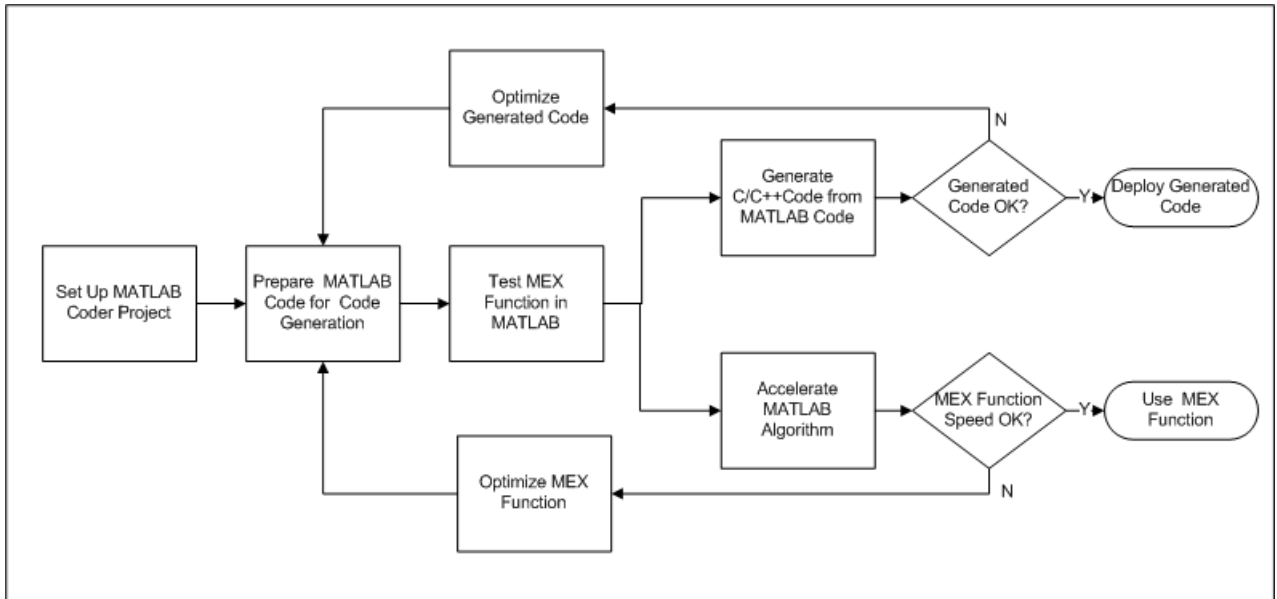
- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize the generated code for a specific target environment.
- Enable tracing options that help you to verify the generated code.
- Generate reusable, reentrant code.

Code Generation for Fixed-Point Algorithms

Using the Fixed-Point Designer™ product, you can generate:

- MEX functions to accelerate fixed-point algorithms.
- Fixed-point code that provides a bit-wise match to MEX function results.

Code Generation Workflow



See Also

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “Code Generation Workflow” on page 19-3
- “Workflow for Accelerating MATLAB Algorithms” on page 22-2

Design Considerations for C/C++ Code Generation

- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Which Code Generation Feature to Use” on page 2-4
- “Prerequisites for C/C++ Code Generation from MATLAB” on page 2-5
- “MATLAB Code Design Considerations for Code Generation” on page 2-6
- “Expected Differences in Behavior After Compiling MATLAB Code” on page 2-8
- “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-12

When to Generate Code from MATLAB Algorithms

Generating code from MATLAB algorithms for desktop and embedded systems allows you to perform your software design, implementation, and testing completely within the MATLAB workspace. You can:

- Verify that your algorithms are suitable for code generation
- Generate efficient, readable, and compact C/C++ code automatically, which eliminates the need to manually translate your MATLAB algorithms and minimizes the risk of introducing errors in the code.
- Modify your design in MATLAB code to take into account the specific requirements of desktop and embedded applications, such as data type management, memory use, and speed.
- Test the generated code and easily verify that your modified algorithms are functionally equivalent to your original MATLAB algorithms.
- Generate MEX functions to:
 - Accelerate MATLAB algorithms in certain applications.
 - Speed up fixed-point MATLAB code.
- Generate hardware description language (HDL) from MATLAB code.

When Not to Generate Code from MATLAB Algorithms

Do not generate code from MATLAB algorithms for the following applications. Use the recommended MathWorks® product instead.

To:	Use:
Deploy an application that uses handle graphics	MATLAB Compiler™
Use Java®	MATLAB Builder™ JA
Use toolbox functions that do not support code generation	Toolbox functions that you rewrite for desktop and embedded applications
Deploy MATLAB based GUI applications on a supported MATLAB host	MATLAB Compiler

To:	Use:
Deploy web-based or Windows® applications	<ul style="list-style-type: none">• MATLAB Builder NE• MATLAB Builder JA
Interface C code with MATLAB	MATLAB mex function

Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	codegen function	MATLAB Coder	Try this in “MEX Function Generation at the Command Line”.
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder user interface	MATLAB Coder	Try this in “C Code Generation Using the Project Interface”.
	codegen function	MATLAB Coder	Try this in “C Code Generation at the Command Line”.
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder user interface	MATLAB Coder	See “Accelerate MATLAB Algorithms” on page 22-11.
	codegen function	MATLAB Coder	
Integrate MATLAB code into Simulink®	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code”.
Speed up fixed-point MATLAB code	fiaccel function	Fixed-Point Designer	Learn more in “Code Acceleration and Code Generation from MATLAB”.
Integrate custom C code into MATLAB and generate efficient, readable code	codegen function	MATLAB Coder	Learn more in “Custom C/C++ Code Integration” on page 21-12.
Integrate custom C code into code generated from MATLAB	coder.ceval function	MATLAB Coder	Learn more in coder.ceval.
Generate HDL from MATLAB code	MATLAB Function block	Simulink and HDL Coder™	Learn more at www.mathworks.com/products/slhdlcoder .

Prerequisites for C/C++ Code Generation from MATLAB

To generate C/C++ or MEX code from MATLAB algorithms, you must install the following software:

- MATLAB Coder product
- C/C++ compiler

MATLAB Code Design Considerations for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

You can choose whether the generated code uses static or dynamic memory allocation.

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get better speed, but with higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- Speed

Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

To improve the speed of the generated code:

- Choose a suitable C/C++ compiler. Do not use the default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms.
- Consider disabling run-time checks.

By default, for safety, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower simulation. Disabling run-time checks usually results in streamlined generated code and faster simulation. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

See Also

- “Data Definition Basics”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data” on page 7-4
- “Control Dynamic Memory Allocation” on page 19-99
- “Control Run-Time Checks” on page 22-18

Expected Differences in Behavior After Compiling MATLAB Code

In this section...
“Why Are There Differences?” on page 2-8
“Character Size” on page 2-8
“Order of Evaluation in Expressions” on page 2-8
“Termination Behavior” on page 2-9
“Size of Variable-Size N-D Arrays” on page 2-9
“Size of Empty Arrays” on page 2-10
“Floating-Point Numerical Results” on page 2-10
“NaN and Infinity Patterns” on page 2-11
“Code Generation Target” on page 2-11
“MATLAB Class Initial Values” on page 2-11
“Variable-Size Support for Code Generation” on page 2-11

Why Are There Differences?

To convert MATLAB code to C/C++ code that works efficiently, the code generation process introduces optimizations that intentionally cause the generated code to behave differently — and sometimes produce different results — from the original source code. This section describes these differences.

Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Code Generation for Characters” on page 6-6.

Order of Evaluation in Expressions

Generated code does not enforce order of evaluation in expressions. For most expressions, order of evaluation is not significant. However, for expressions

with side effects, the generated code may produce the side effects in different order from the original MATLAB code. Expressions that produce side effects include those that:

- Modify persistent or global variables
- Display data to the screen
- Write data to files
- Modify the properties of handle class objects

In addition, the generated code does not enforce order of evaluation of logical operators that do not short circuit.

For more predictable results, it is good coding practice to split expressions that depend on the order of evaluation into multiple statements. For example, rewrite:

```
A = f1() + f2();
```

as

```
A = f1();  
A = A + f2();
```

so that the generated code calls `f1` before `f2`.

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, optimizations remove infinite loops from generated code if they do not have side effects. As a result, the generated code may terminate even though the corresponding MATLAB code does not.

Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function might return a different result in generated code than in MATLAB source code. The `size` function sometimes returns trailing ones (singleton dimensions) in generated code, but always drops trailing ones in MATLAB. For example, for an N-D array `X` with dimensions `[4 2 1 1]`, `size(X)` might return `[4 2 1 1]` in generated code,

but always returns [4 2] in MATLAB. See “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-29.

Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. See “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 7-30.

Floating-Point Numerical Results

The generated code might not produce the same floating-point numerical results as MATLAB in the following situations:

When computer hardware uses extended precision registers

Results vary depending on how the C/C++ compiler allocates extended precision floating-point registers. Computation results might not match MATLAB calculations because of different compiler optimization settings or different code surrounding the floating-point calculations.

For certain advanced library functions

The generated code might use different algorithms to implement certain advanced library functions, such as `fft`, `svd`, `eig`, `mldivide`, and `mrdivide`.

For example, the generated code uses a simpler algorithm to implement `svd` to accommodate a smaller footprint. Results might also vary according to matrix properties. For example, MATLAB might detect symmetric or Hermitian matrices at run time and switch to specialized algorithms that perform computations faster than implementations in the generated code.

For implementation of BLAS library functions

For implementations of BLAS library functions. Generated C/C++ code uses reference implementations of BLAS functions, which may produce different results from platform-specific BLAS implementations in MATLAB.

NaN and Infinity Patterns

The generated code might not produce exactly the same pattern of NaN and inf values as MATLAB code when these values are mathematically meaningless. For example, if MATLAB output contains a NaN, output from the generated code should also contain a NaN, but not necessarily in the same place.

Code Generation Target

The `coder.target` function returns different values in MATLAB than in the generated code. The intent is to help you determine whether your function is executing in MATLAB or has been compiled for a simulation or code generation target. See `coder.target`.

MATLAB Class Initial Values

MATLAB computes class initial values at class loading time before code generation. The code generation software uses the value that MATLAB computed, it does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generation software does not execute this function. If the function modifies a global state, for example, a persistent variable, code generation software might provide a different initial value than MATLAB. For more information, see “Defining Class Properties for Code Generation” on page 10-4.

Variable-Size Support for Code Generation

For incompatibilities with MATLAB in variable-size support for code generation, see:

- “Incompatibility with MATLAB for Scalar Expansion” on page 7-27
- “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-29
- “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 7-30
- “Incompatibility with MATLAB in Vector-Vector Indexing” on page 7-31
- “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-32

MATLAB Language Features Supported for C/C++ Code Generation

MATLAB supports the following language features in generated code:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “Variable-Size Data Definition for Code Generation” on page 7-3)
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-32)
- Complex numbers (see “Code Generation for Complex Data” on page 6-4)
- Numeric classes (see “Supported Variable Types” on page 5-18)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB”)
- Program control statements `if`, `switch`, `for`, and `while`
- Arithmetic, relational, and logical operators
- Local functions
- Persistent variables (see “Define and Initialize Persistent Variables” on page 5-10)
- Global variables (see “Specifying Global Variable Type and Initial Value in a Project” on page 16-33).
- Structures
- Characters (see “Code Generation for Characters” on page 6-6)
- Function handles
- Frames
- Variable length input and output argument lists
- Subset of MATLAB toolbox functions
- MATLAB classes

- Ability to call functions (see “Resolution of Function Calls in MATLAB Generated Code” on page 13-2)

MATLAB Language Features Not Supported for C/C++ Code Generation

MATLAB does not support the following features in generated code:

- Anonymous functions
- Cell arrays
- Java
- Nested functions
- Recursion
- Sparse matrices
- try/catch statements

System Objects Supported for Code Generation

System Objects Supported for Code Generation

In this section...
“Code Generation for System Objects” on page 3-2
“Computer Vision System Toolbox System Objects” on page 3-2
“Communications System Toolbox System Objects” on page 3-7
“DSP System Toolbox System Objects” on page 3-13

Code Generation for System Objects

You can generate C/C++ code for a subset of System objects provided by Communications System Toolbox, DSP System Toolbox, and Computer Vision System Toolbox. To use these System objects, you need to install the requisite toolbox.

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets. For general information on MATLAB objects, see “Begin Using Object-Oriented Programming”.

Computer Vision System Toolbox System Objects

If you install Computer Vision System Toolbox software, you can generate C/C++ code for the following Computer Vision System Toolbox System objects. For more information on how to use these System objects, see “System Objects in MATLAB Code Generation”.

Supported Computer Vision System Toolbox System Objects

Object	Description
Analysis & Enhancement	
vision.BoundaryTracer	Trace object boundaries in binary images.
vision.ContrastAdjuster	Adjust image contrast by linear scaling.
vision.Deinterlacer	Remove motion artifacts by deinterlacing input video signal.
vision.EdgeDetector	Find edges of objects in images.
vision.ForegroundDetector	Detect foreground using Gaussian Mixture Models. This object supports tunable properties in code generation.
vision.HistogramEqualizer	Enhance contrast of images using histogram equalization.
vision.TemplateMatcher	Perform template matching by shifting template over image.
Conversions	
vision.Autothresher	Convert intensity image to binary image.
vision.ChromaResampler	Downsample or upsample chrominance components of images.
vision.ColorSpaceConverter	Convert color information between color spaces.
vision.DemosaicInterpolator	Demosaic Bayer's format images.
vision.GammaCorrector	Apply or remove gamma correction from images or video streams.
vision.ImageComplementer	Compute complement of pixel values in binary, intensity, or RGB images.
vision.ImageDataTypeConverter	Convert and scale input image to specified output data type.
Feature Detection, Extraction, and Matching	

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
vision.CornerDetector	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
vision.Convolver	Compute 2-D discrete convolution of two input matrices.
vision.ImageFilter	Perform 2-D FIR filtering of input matrix.
vision.MedianFilter	2D median filtering.
Geometric Transformations	
vision.GeometricRotator	Rotate image by specified angle.
vision.GeometricScaler	Enlarge or shrink image size.
vision.GeometricShearer	Shift rows or columns of image by linearly varying offset.
vision.GeometricTransformer	Apply projective or affine transformation to an image.
vision.GeometricTransformEstimator	Estimate geometric transformation from matching point pairs.
vision.GeometricTranslator	Translate image in two-dimensional plane using displacement vector.
Morphological Operations	
vision.ConnectedComponentLabeler	Label and count the connected regions in a binary image.
vision.MorphologicalClose	Perform morphological closing on image.
vision.MorphologicalDilate	Perform morphological dilation on an image.
vision.MorphologicalErode	Perform morphological erosion on an image.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
vision.MorphologicalOpen	Perform morphological opening on an image.
Object Detection	
vision.HistogramBasedTracker	Track object in video based on histogram. This object supports tunable properties in code generation.
Sinks	
vision.VideoPlayer	Send video data to computer screen. This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable.
vision.DeployableVideoPlayer	Send video data to computer screen.
vision.VideoFileWriter	Write video frames and audio samples to multimedia file.
Sources	
vision.VideoFileReader	Read video frames and audio samples from compressed multimedia file.
Statistics	
vision.Autocorrelator	Compute 2-D autocorrelation of input matrix.
vision.BlobAnalysis	Compute statistics for connected regions in a binary image.
vision.Crosscorrelator	Compute 2-D cross-correlation of two input matrices.
vision.Histogram	Generate histogram of each input matrix. This object has no tunable properties.
vision.LocalMaximaFinder	Find local maxima in matrices.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
vision.Maximum	Find maximum values in input or sequence of inputs.
vision.Mean	Find mean value of input or sequence of inputs.
vision.Median	Find median values in an input.
vision.Minimum	Find minimum values in input or sequence of inputs.
vision.PSNR	Compute peak signal-to-noise ratio (PSNR) between images.
vision.StandardDeviation	Find standard deviation of input or sequence of inputs.
vision.Variance	Find variance values in an input or sequence of inputs.
Text & Graphics	
vision.AlphaBlender	Combine images, overlay images, or highlight selected pixels.
vision.MarkerInserter	Draw markers on output image.
vision.ShapeInserter	Draw rectangles, lines, polygons, or circles on images.
vision.TextInserter	Draw text on image or video stream.
Transforms	
vision.DCT	Compute 2-D discrete cosine transform.
vision.FFT	Two-dimensional discrete Fourier transform.
vision.HoughLines	Find Cartesian coordinates of lines that are described by rho and theta pairs.
vision.HoughTransform	Find lines in images via Hough transform.
vision.IDCT	Compute 2-D inverse discrete cosine transform.
vision.IFFT	Two-dimensional inverse discrete Fourier transform.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
vision.Pyramid	Perform Gaussian pyramid decomposition.
Utilities	
vision.ImagePadder	Pad or crop input image along its rows, columns, or both.

Communications System Toolbox System Objects

If you install Communications System Toolbox software, you can generate C/C++ code for the following Communications System Toolbox System objects. For information on how to use these System objects, see “Code Generation with System Objects”.

Supported Communications System Toolbox System Objects

Object	Description
Source Coding	
comm.DifferentialDecoder	Decode binary signal using differential decoding
comm.DifferentialEncoder	Encode binary signal using differential coding
Channels	
comm.AWGNChannel	Add white Gaussian noise to input signal
comm.LTEMIMOChannel	Filter input signal through LTE MIMO multipath fading channel
comm.MIMOChannel	Filter input signal through MIMO multipath fading channel
comm.BinarySymmetricChannel	Introduce binary errors
Equalizers	
comm.MLSEEqualizer	Equalize using maximum likelihood sequence estimation
Filters	

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.IntegrateAndDumpFilter	Integrate discrete-time signal with periodic resets
Measurements	
comm.ACPR	Measure adjacent channel power ratio
comm.CCDF	Measure complementary cumulative distribution function
comm.EVM	Measure error vector magnitude
comm.MER	Measure modulation error ratio
Sources	
comm.BarkerCode	Generate Barker code
comm.HadamardCode	Generate Hadamard code
comm.KasamiSequence	Generate a Kasami sequence
comm.OVSFCode	Generate OVSF code
comm.PNSequence	Generate a pseudo-noise (PN) sequence
comm.WalshCode	Generate Walsh code from orthogonal set of codes
Error Detection and Correction – Block Coding	
comm.BCHDecoder	Decode data using BCH decoder
comm.BCHEncoder	Encode data using BCH encoder
comm.LDPCDecoder	Decode binary low-density parity-check code
comm.LDPCEncoder	Encode binary low-density parity-check code
comm.RSDecoder	Decode data using Reed-Solomon decoder
comm.RSEncoder	Encode data using Reed-Solomon encoder
Error Detection and Correction – Convolutional Coding	
comm.ConvolutionalEncoder	Convolutionally encode binary data
comm.ViterbiDecoder	Decode convolutionally encoded data using Viterbi algorithm

Supported Communications System Toolbox System Objects (Continued)

Object	Description
Error Detection and Correction – Cyclic Redundancy Check Coding	
comm.CRCDetector	Detect errors in input data using cyclic redundancy code
comm.CRCGenerator	Generate cyclic redundancy code bits and append to input data
comm.HDLCRCGenerator	Generate CRC code bits and append to input data, optimized for HDL code generation
comm.TurboDecoder	Decode input signal using parallel concatenated decoding scheme
comm.TurboEncoder	Encode input signal using parallel concatenated encoding scheme
Interleavers – Block	
comm.AlgebraicDeinterleaver	Deinterleave input symbols using algebraically derived permutation vector
comm.AlgebraicInterleaver	Permute input symbols using an algebraically derived permutation vector
comm.BlockDeinterleaver	Deinterleave input symbols using permutation vector
comm.BlockInterleaver	Permute input symbols using a permutation vector
comm.MatrixDeinterleaver	Deinterleave input symbols using permutation matrix
comm.MatrixInterleaver	Permute input symbols using permutation matrix
comm.MatrixHelicalScanDeinterleaver	Deinterleave input symbols by filling a matrix along diagonals
comm.MatrixHelicalScanInterleaver	Permute input symbols by selecting matrix elements along diagonals
Interleavers – Convolutional	
comm.ConvolutionalDeinterleaver	Restore ordering of symbols using shift registers

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.ConvolutionalInterleaver	Permute input symbols using shift registers
comm.HelicalDeinterleaver	Restore ordering of symbols using a helical array
comm.HelicalInterleaver	Permute input symbols using a helical array
comm.MultiplexedDeinterleaver	Restore ordering of symbols using a set of shift registers with specified delays
comm.MultiplexedInterleaver	Permute input symbols using a set of shift registers with specified delays
MIMO	
comm.OSTBCCombiner	Combine inputs using orthogonal space-time block code
comm.OSTBCEncoder	Encode input message using orthogonal space-time block code
Digital Baseband Modulation – Phase	
comm.BPSKDemodulator	Demodulate using binary PSK method
comm.BPSKModulator	Modulate using binary PSK method
comm.DBPSKModulator	Modulate using differential binary PSK method
comm.DPSKDemodulator	Demodulate using M-ary DPSK method
comm.DPSKModulator	Modulate using M-ary DPSK method
comm.DQPSKDemodulator	Demodulate using differential quadrature PSK method
comm.DQPSKModulator	Modulate using differential quadrature PSK method
comm.DBPSKDemodulator	Demodulate using M-ary DPSK method
comm.QPSKDemodulator	Demodulate using quadrature PSK method
comm.QPSKModulator	Modulate using quadrature PSK method
comm.PSKDemodulator	Demodulate using M-ary PSK method
comm.PSKModulator	Modulate using M-ary PSK method

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.OQPSKDemodulator	Demodulate offset quadrature PSK modulated data
comm.OQPSKModulator	Modulate using offset quadrature PSK method
Digital Baseband Modulation – Amplitude	
comm.GeneralQAMDemodulator	Demodulate using arbitrary QAM constellation. This object has no tunable properties in code generation.
comm.GeneralQAMModulator	Modulate using arbitrary QAM constellation
comm.PAMDemodulator	Demodulate using M-ary PAM method
comm.PAMModulator	Modulate using M-ary PAM method
comm.RectangularQAMDemodulator	Demodulate using rectangular QAM method
comm.RectangularQAMModulator	Modulate using rectangular QAM method
Digital Baseband Modulation – Frequency	
comm.FSKDemodulator	Demodulate using M-ary FSK method
comm.FSKModulator	Modulate using M-ary FSK method
Digital Baseband Modulation – Trellis Coded	
comm.GeneralQAMTCMDemodulator	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation
comm.GeneralQAMTCMModulator	Convolutionally encode binary data and map using arbitrary QAM constellation
comm.PSKTCMDemodulator	Demodulate convolutionally encoded data mapped to M-ary PSK constellation
comm.PSKTCMModulator	Convolutionally encode binary data and map using M-ary PSK constellation
comm.RectangularQAMTCMDemodulator	Demodulate convolutionally encoded data mapped to rectangular QAM constellation
comm.RectangularQAMTCMModulator	Convolutionally encode binary data and map using rectangular QAM constellation

Supported Communications System Toolbox System Objects (Continued)

Object	Description
Digital Baseband Modulation – Continuous Phase	
comm.CPFSKDemodulator	Demodulate using CPFSK method and Viterbi algorithm
comm.CPFSKModulator	Modulate using CPFSK method
comm.CPMDemodulator	Demodulate using CPM method and Viterbi algorithm
comm.CPMModulator	Modulate using CPM method
comm.GMSKDemodulator	Demodulate using GMSK method and the Viterbi algorithm
comm.GMSKModulator	Modulate using GMSK method
comm.MSKDemodulator	Demodulate using MSK method and the Viterbi algorithm
comm.MSKModulator	Modulate using MSK method
RF Impairments	
comm.MemorylessNonlinearity	Apply memoryless nonlinearity to input signal
comm.PhaseFrequencyOffset	Apply phase and frequency offsets to input signal. The PhaseOffset property of this object is not tunable in code generation.
comm.PhaseNoise	Apply phase noise to complex baseband signal
comm.ThermalNoise	Add receiver thermal noise
Synchronization – Timing Phase	
comm.EarlyLateGateTimingSynchronizer	Recover symbol timing phase using early-late gate method
comm.GardnerTimingSynchronizer	Recover symbol timing phase using Gardner's method
comm.GMSKTimingSynchronizer	Recover symbol timing phase using fourth-order nonlinearity method

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.MSKTimingSynchronizer	Recover symbol timing phase using fourth-order nonlinearity method
comm.MuellerMullerTimingSynchronizer	Recover symbol timing phase using Mueller-Muller method
Synchronization Utilities	
comm.CPMCarrierPhaseSynchronizer	Recover carrier phase of baseband CPM signal
comm.DiscreteTimeVCO	Generate variable frequency sinusoid
Converters	
comm.BitToInteger	Convert vector of bits to vector of integers
comm.IntegerToBit	Convert vector of integers to vector of bits
Sequence Operators	
comm.Descrambler	Descramble input signal
comm.GoldSequence	Generate Gold sequence
comm.Scrambler	Scramble input signal

DSP System Toolbox System Objects

If you install DSP System Toolbox software, you can generate C/C++ code for the following DSP System Toolbox System objects. For information on how to use these System objects, see “Code Generation with System Objects”.

Supported DSP System Toolbox System Objects

Object	Description
Estimation	
dsp.BurgAREstimator	Compute estimate of autoregressive model parameters using Burg method
dsp.BurgSpectrumEstimator	Compute parametric spectral estimate using Burg method
dsp.CepstralToLPC	Convert cepstral coefficients to linear prediction coefficients
dsp.LevinsonSolver	Solve linear system of equations using Levinson-Durbin recursion
dsp.LPCToAutocorrelation	Convert linear prediction coefficients to autocorrelation coefficients
dsp.LPCToCepstral	Convert linear prediction coefficients to cepstral coefficients
dsp.LPCToLSF	Convert linear prediction coefficients to line spectral frequencies
dsp.LPCToLSP	Convert linear prediction coefficients to line spectral pairs
dsp.LPCToRC	Convert linear prediction coefficients to reflection coefficients
dsp.LSFToLPC	Convert line spectral frequencies to linear prediction coefficients
dsp.LSPToLPC	Convert line spectral pairs to linear prediction coefficients
dsp.RCToAutocorrelation	Convert reflection coefficients to autocorrelation coefficients
dsp.RCToLPC	Convert reflection coefficients to linear prediction coefficients
Filters	
dsp.AffineProjectionFilter	Adaptive filter using the Affine Projection algorithm

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.AllpoleFilter</code>	IIR Filter with no zeros. Only the Denominator property is tunable for code generation.
<code>dsp.BiquadFilter</code>	Model biquadratic IIR (SOS) filters
<code>dsp.CICDecimator</code>	Decimate input using Cascaded Integrator-Comb filter
<code>dsp.CICInterpolator</code>	Interpolate signal using Cascaded Integrator-Comb filter
<code>dsp.DigitalFilter</code>	Filter each channel of input over time using discrete-time filter implementations. The <code>SOSMatrix</code> and <code>ScaleValues</code> properties are not supported for code generation.
<code>dsp.FIRDecimator</code>	Filter and downsample input signals
<code>dsp.FIRFilter</code>	Static or time-varying FIR filter. Only the Numerator property is tunable for code generation.
<code>dsp.FIRInterpolator</code>	Upsample and filter input signals
<code>dsp.FIRRateConverter</code>	Upsample, filter and downsample input signals
<code>dsp.IIRFilter</code>	Infinite Impulse Response (IIR) filter. Only the Numerator and Denominator properties are tunable for code generation.
<code>dsp.LMSFilter</code>	Compute output, error, and weights using LMS adaptive algorithm
<code>dsp.RLSFilter</code>	Adaptive filter using the Recursive Least Squares (RLS) algorithm
Math Operations	
<code>dsp.ArrayVectorAdder</code>	Add vector to array along specified dimension
<code>dsp.ArrayVectorDivider</code>	Divide array by vector along specified dimension
<code>dsp.ArrayVectorMultiplier</code>	Multiply array by vector along specified dimension

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.ArrayVectorSubtractor</code>	Subtract vector from array along specified dimension
<code>dsp.CumulativeProduct</code>	Compute cumulative product of channel, column, or row elements
<code>dsp.CumulativeSum</code>	Compute cumulative sum of channel, column, or row elements
<code>dsp.LDLFactor</code>	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion
<code>dsp.LowerTriangularSolver</code>	Solve $LX = B$ for X when L is lower triangular matrix
<code>dsp.LUFactor</code>	Factor square matrix into lower and upper triangular matrices
<code>dsp.Normalizer</code>	Normalize input
<code>dsp.UpperTriangularSolver</code>	Solve $UX = B$ for X when U is upper triangular matrix
Quantizers	
<code>dsp.ScalarQuantizerDecoder</code>	Convert each index value into quantized output value
<code>dsp.ScalarQuantizerEncoder</code>	Perform scalar quantization encoding
<code>dsp.VectorQuantizerDecoder</code>	Find vector quantizer codeword for given index value
<code>dsp.VectorQuantizerEncoder</code>	Perform vector quantization encoding
Scopes	
<code>dsp.SpectrumAnalyzer</code>	Display frequency spectrum of time-domain signals. This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
<code>dsp.TimeScope</code>	Display time-domain signals. This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
Signal Management	

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.Counter	Count up or down through specified range of numbers
dsp.DelayLine	Rebuffer sequence of inputs with one-sample shift
Signal Operations	
dsp.Convolver	Compute convolution of two inputs
dsp.Delay	Delay input by specified number of samples or frames
dsp.Interpolator	Interpolate values of real input samples
dsp.NCO	Generate real or complex sinusoidal signals
dsp.PeakFinder	Determine extrema (maxima or minima) in input signal
dsp.PhaseUnwrapper	Unwrap signal phase
dsp.VariableFractionalDelay	Delay input by time-varying fractional number of sample periods
dsp.VariableIntegerDelay	Delay input by time-varying integer number of sample periods
dsp.Window	Generate or apply window function. This object has no tunable properties for code generation.
dsp.ZeroCrossingDetector	Calculate number of zero crossings of a signal
Sinks	
dsp.AudioPlayer	Write audio data to computer's audio device
dsp.AudioFileWriter	Write audio file
dsp.UDPsender	Send UDP packets to the network
Sources	
dsp.AudioFileReader	Read audio samples from an audio file
dsp.AudioRecorder	Read audio data from computer's audio device
dsp.SignalSource	Import variable from workspace

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.SineWave	Generate discrete sine wave. This object has no tunable properties for code generation.
dsp.UDPReceiver	Receive UDP packets from the network
Statistics	
dsp.Autocorrelator	Compute autocorrelation of vector inputs
dsp.Crosscorrelator	Compute cross-correlation of two inputs
dsp.Histogram	Output histogram of an input or sequence of inputs. This object has no tunable properties for code generation.
dsp.Maximum	Compute maximum value in input
dsp.Mean	Compute average or mean value in input
dsp.Median	Compute median value in input
dsp.Minimum	Compute minimum value in input
dsp.RMS	Compute root-mean-square of vector elements
dsp.StandardDeviation	Compute standard deviation of vector elements
dsp.Variance	Compute variance of input or sequence of inputs
Transforms	
dsp.AnalyticSignal	Compute analytic signals of discrete-time inputs
dsp.DCT	Compute discrete cosine transform (DCT) of input
dsp.FFT	Compute fast Fourier transform (FFT) of input
dsp.IDCT	Compute inverse discrete cosine transform (IDCT) of input
dsp.IFFT	Compute inverse fast Fourier transform (IFFT) of input

Functions Supported for Code Generation

- “Functions Supported for Code Generation — Alphabetical List” on page 4-2
- “Functions Supported for Code Generation — Categorical List” on page 4-79

Functions Supported for Code Generation – Alphabetical List

You can generate efficient C/C++ code for a subset of MATLAB and toolbox functions that you call from MATLAB code. In generated code, each supported function has the same name, arguments, and functionality as its MATLAB or toolbox counterparts. However, to generate code for these functions, you must adhere to certain limitations when calling them from your MATLAB source code. These limitations appear in the list below.

To find supported functions by MATLAB category or toolbox, see “Functions Supported for Code Generation — Categorical List” on page 4-79.

Note For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB”.

Function	Product	Remarks/Limitations
abs	MATLAB	—
abs	Fixed-Point Designer	—
accumneg	Fixed-Point Designer	—
accumpos	Fixed-Point Designer	—
acos	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
acosd	MATLAB	—

Function	Product	Remarks/Limitations
acosh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
add	Fixed-Point Designer	—
all	MATLAB	—
all	Fixed-Point Designer	—
and	MATLAB	—
angle	MATLAB	—
any	MATLAB	—
any	Fixed-Point Designer	—
asec	MATLAB	—
asecd	MATLAB	—
asech	MATLAB	—

Function	Product	Remarks/Limitations
asin	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
asind	MATLAB	—
asinh	MATLAB	—
assert	MATLAB	<ul style="list-style-type: none"> Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time.
atan	MATLAB	—
atan2	MATLAB	—
atan2	Fixed-Point Designer	—
atan2d	MATLAB	—
atand	MATLAB	—
atanh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.

Function	Product	Remarks/Limitations
barthannwin	Signal Processing Toolbox™	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
bartlett	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
besselap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter order must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
beta	MATLAB	—
betainc	MATLAB	—
betaln	MATLAB	—
bi2de	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
bin2dec	MATLAB	<ul style="list-style-type: none"> Does not match MATLAB when the input is empty.
binaryFeatures	Computer Vision System Toolbox	—
bitand	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitand	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.

Function	Product	Remarks/Limitations
bitandreduce	Fixed-Point Designer	—
bitcmp	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitcmp	Fixed-Point Designer	—
bitconcat	Fixed-Point Designer	—
bitget	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitget	Fixed-Point Designer	—
bitmax	MATLAB	—
bitor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitor	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	Fixed-Point Designer	—
bitreplicate	Fixed-Point Designer	—
bitrevorder	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Computation performed at run time.

Function	Product	Remarks/Limitations
bitrol	Fixed-Point Designer	—
bitror	Fixed-Point Designer	—
bitset	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitset	Fixed-Point Designer	—
bitshift	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitshift	Fixed-Point Designer	—
bitsliceget	Fixed-Point Designer	—
bitsll	Fixed-Point Designer	—
bitsra	Fixed-Point Designer	—
bitsrl	Fixed-Point Designer	—
bitxor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitxor	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
blackman	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
blackmanharris	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
blanks	MATLAB	—
blkdiag	MATLAB	—

Function	Product	Remarks/Limitations
bohmanwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
bsxfun	MATLAB	—
buttap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter order must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
butter	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
buttord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
bwlookup	Image Processing Toolbox™	<ul style="list-style-type: none"> • For best results, specify an input image of class <code>logical</code>.
bwmorph	Image Processing Toolbox	<ul style="list-style-type: none"> • The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code>.
cart2pol	MATLAB	—

Function	Product	Remarks/Limitations
cart2sph	MATLAB	—
cast	MATLAB	—
cat	MATLAB	—
ceil	MATLAB	—
ceil	Fixed-Point Designer	—
cfirpm	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
char	MATLAB	—
cheb1ap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
cheb1ord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
cheb2ap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
cheb2ord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
chebwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
cheby1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
cheby2	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
chol	MATLAB	<ul style="list-style-type: none"> When there are two output arguments, either make the input matrix variable-size in both dimensions, or, if the input matrix must be fixed size, copy the input matrix to a variable-size matrix before calling chol. <pre> coder.varsize('B'); B = A; [B,ρ] = chol(B); </pre>
circshift	MATLAB	—
class	MATLAB	—
compan	MATLAB	—
complex	MATLAB	—
complex	Fixed-Point Designer	—
computer	MATLAB	<ul style="list-style-type: none"> Information about the computer on which the code generation software is running. Use only when the code generation target is S-function (Simulation) or MEX-function.
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Designer	—
conndef	Image Processing Toolbox	All input arguments must be compile-time constants.
conv	MATLAB	—

Function	Product	Remarks/Limitations
conv	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. • For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> ▪ In generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. ▪ In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
conv2	MATLAB	—
convergent	Fixed-Point Designer	—
convn	MATLAB	—
cordicabs	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicangle	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicatan2	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordiccart2pol	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.

Function	Product	Remarks/Limitations
cordiccxep	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordiccos	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicpol2cart	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicrotate	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicsin	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicsincos	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-size signals are not supported.
corrcoef	MATLAB	<ul style="list-style-type: none"> Row-vector input is only supported when the first two inputs are vectors and nonscalar.
cos	MATLAB	—
cos	Fixed-Point Designer	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	—
coth	MATLAB	—
cov	MATLAB	—
cross	MATLAB	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant.
csc	MATLAB	—
cscd	MATLAB	—
csch	MATLAB	—
ctranspose	MATLAB	—

Function	Product	Remarks/Limitations
ctranspose	Fixed-Point Designer	—
cumprod	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to <code>double</code> first.
cumsum	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to <code>double</code> first.
cumtrapz	MATLAB	—
dct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
de2bi	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
deal	MATLAB	—
deblank	MATLAB	<ul style="list-style-type: none"> Supports only inputs from the <code>char</code> class. Input values must be in the range 0-127.

Function	Product	Remarks/Limitations
dec2bin	MATLAB	<ul style="list-style-type: none"> • If input <i>d</i> is double, <i>d</i> must be less than 2^{52}. • If input <i>d</i> is single, <i>d</i> must be less than 2^{23}. • Unless you specify input <i>n</i> to be constant and <i>n</i> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <i>n</i> must be at least 52 for double, 23 for single, 16 for char, 32 for int32, 16 for int16, and so on.
dec2hex	MATLAB	<ul style="list-style-type: none"> • If input <i>d</i> is double, <i>d</i> must be less than 2^{52}. • If input <i>d</i> is single, <i>d</i> must be less than 2^{23}. • Unless you specify input <i>n</i> to be constant and <i>n</i> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <i>n</i> must be at least 13 for double, 6 for single, 4 for char, 8 for int32, 4 for int16, and so on.
deconv	MATLAB	—
de12	MATLAB	—
det	MATLAB	—

Function	Product	Remarks/Limitations
detrnd	MATLAB	<ul style="list-style-type: none"> • If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> ▪ Be real ▪ Be sorted in ascending order ▪ Restrict elements to integers in the interval $[1, n-2]$, where n is the number of elements in a column of input argument <code>X</code>, or the number of elements in <code>X</code> when <code>X</code> is a row vector ▪ Contain all unique values
diag	MATLAB	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
diag	Fixed-Point Designer	<ul style="list-style-type: none"> • If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
diff	MATLAB	<ul style="list-style-type: none"> • If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.
divide	Fixed-Point Designer	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Complex and imaginary divisors are not supported. • The syntax <code>T.divide(a,b)</code> is not supported.

Function	Product	Remarks/Limitations
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Designer	—
downsample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs.
dpss	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
eig	MATLAB	<ul style="list-style-type: none"> QZ algorithm used in all cases, whereas MATLAB might use different algorithms for different inputs. Consequently, V might represent a different basis of eigenvectors, and the eigenvalues in D might not be in the same order as in MATLAB. With one input, $[V,D] = \text{eig}(A)$, the results will be similar to those obtained using $[V,D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')$ in MATLAB, except that for code generation, the columns of V are normalized.

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Options 'balance', 'nobalance' are not supported for the standard eigenvalue problem, and 'chol' is not supported for the symmetric generalized eigenvalue problem. Outputs are of complex type.
ellip	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
ellipap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
ellipke	MATLAB	—

Function	Product	Remarks/Limitations
ellipord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
end	Fixed-Point Designer	—
epipolarLine	Computer Vision System Toolbox	—
eps	MATLAB	—
eps	Fixed-Point Designer	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi single</code> and <code>fi double</code> signals.
eq	MATLAB	—
eq	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—

Function	Product	Remarks/Limitations
error	MATLAB	<ul style="list-style-type: none"> This is an extrinsic call.
estimateFundamentalMatrix	Computer Vision System Toolbox	—
estimateUncalibratedRectification	Computer Vision System Toolbox	—
exp	MATLAB	—
expint	MATLAB	—
expm	MATLAB	—
expm1	MATLAB	—
extractFeatures	Computer Vision System Toolbox	—
eye	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
factor	MATLAB	<ul style="list-style-type: none"> For double precision input, the maximum value of A is $2^{32} - 1$. For single precision input, the maximum value of A is $2^{24} - 1$.
factorial	MATLAB	—
false	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
fclose	MATLAB	—
fft	MATLAB	<ul style="list-style-type: none"> Length of input vector must be a power of 2.
fft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.
fftn	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.

Function	Product	Remarks/Limitations
fftshift	MATLAB	—
fi	Fixed-Point Designer	<ul style="list-style-type: none"> • Use to create a fixed-point constant or variable. • The default constructor syntax without input arguments is not supported. • The syntax <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v, 'PropertyName',PropertyValue...)</code>. • If the input value is not known at compile time, you must provide complete <code>numericType</code> information. • <code>numericType</code> object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
filter2	MATLAB	—

Function	Product	Remarks/Limitations
filtfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use coder.Constant. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
fimath	Fixed-Point Designer	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned the fimath object defined in the MATLAB Function dialog in the Model Explorer. Use to create fimath objects in generated code.
find	MATLAB	<ul style="list-style-type: none"> Issues an error if a variable-sized input becomes a row vector at run time. <hr/> <p>Note This limitation does not apply when the input is scalar or a variable-length row vector.</p> <hr/> <ul style="list-style-type: none"> For variable-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or [] at run

Function	Product	Remarks/Limitations
		<p>time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.</p>
fir1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
fir2	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
fircls	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
fircls1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
firls	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
firpm	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
firpmord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
firrcos	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
fix	MATLAB	—
fix	Fixed-Point Designer	—
fixed.Quantizer	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
flattopwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
flintmax	MATLAB	—
flipdim	MATLAB	—
fliplr	MATLAB	—
flipud	MATLAB	—
floor	MATLAB	—
floor	Fixed-Point Designer	—
fopen	MATLAB	<ul style="list-style-type: none"> Does not support: <ul style="list-style-type: none"> machineformat, encoding, or fileID inputs message output fopen('all') If you disable extrinsic calls, you cannot return fileIDs created with fopen to MATLAB or extrinsic functions. You can only use such fileIDs internally.

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> You can open up to 18 files when generating C/C++ executables, static libraries, or dynamic libraries.
fprintf	MATLAB	<ul style="list-style-type: none"> Does not support <ul style="list-style-type: none"> b and t subtypes on %u, %o %x, and %X formats \$ flag for reusing input arguments printing arrays There is no automatic casting. Input arguments must match their format types for predictable results. Escaped characters are limited to the decimal range of 0 – 127. When you disable extrinsic functions in MATLAB, Simulink, or parfor loops, then a call to fprintf with fileID equal to 1 or 2 becomes printf in the generated C/C++ code.
freqspace	MATLAB	—
freqz	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See “freqz With No Output Arguments”.
fspecial	Image Processing Toolbox	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
full	MATLAB	—

Function	Product	Remarks/Limitations
fzero	MATLAB	<ul style="list-style-type: none"> • The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument. • Supports up to three output arguments. Does not support the fourth output argument (the output structure). • Only supports the TolX and FunValCheck fields of an options input structure. Ignores other options in an options input structure. You cannot use the optimset function to create the options structure. Create this structure directly, for example, <pre style="margin-left: 20px;">opt.TolX = tol; opt.FunValCheck = 'on';</pre> <p>The input structure field names must match exactly.</p>
gamma	MATLAB	—
gammainc	MATLAB	—
gammaIn	MATLAB	—

Function	Product	Remarks/Limitations
gaussfir	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
gausswin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
get	Fixed-Point Designer	<ul style="list-style-type: none"> The syntax <code>structure = get(o)</code> is not supported.
getlsb	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
getmsb	Fixed-Point Designer	—
gradient	MATLAB	—
gt	MATLAB	—
gt	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
hadamard	MATLAB	—
hamming	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
hankel	MATLAB	—
hann	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>

Function	Product	Remarks/Limitations
hdlram	Fixed-Point Designer	—
hex2dec	MATLAB	—
hex2num	MATLAB	<ul style="list-style-type: none"> For $n = \text{hex2num}(S)$, $\text{size}(S,2) \leq \text{length}(\text{num2hex}(0))$
hilb	MATLAB	—
hist	MATLAB	<ul style="list-style-type: none"> Histogram bar plotting not supported; call with at least one output argument. If supplied, the second argument x must be a scalar constant. Inputs must be real.
histc	MATLAB	<ul style="list-style-type: none"> The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.
horzcat	Fixed-Point Designer	—
hypot	MATLAB	—
idct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify</p>

Function	Product	Remarks/Limitations
		Constant Inputs at the Command Line” on page 19-46.
idivide	MATLAB	<ul style="list-style-type: none"> For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.
ifft	MATLAB	<ul style="list-style-type: none"> Length of input vector must be a power of 2. Output of ifft block is complex. Does not support the 'symmetric' option.
ifft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2. Does not support the 'symmetric' option.
ifftn	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2. Does not support the 'symmetric' option.
ifftshift	MATLAB	—
imag	MATLAB	—
imag	Fixed-Point Designer	—
imcomplement	Image Processing Toolbox	Does not support int64 and uint64 data types.

Function	Product	Remarks/Limitations
imfill	Image Processing Toolbox	<p>The optional input connectivity, <code>conn</code>, and the string 'holes' must be compile-time constants.</p> <p>Supports only up to 3-D inputs. (No N-D support.)</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code>, is not supported in code generation.</p> <p><code>locations</code> can be a P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-by-ndims(I) matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile time, you cannot change it at run time. However, the number of points in <code>locations</code> can be varied at run time.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmax	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmin	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Function	Product	Remarks/Limitations
imreconstruct	Image Processing Toolbox	The optional third input argument, <code>conn</code> , must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
imregionalmax	Image Processing Toolbox	The optional second input argument, <code>conn</code> , must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
imregionalmin	Image Processing Toolbox	The optional second input argument, <code>conn</code> , must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
ind2sub	MATLAB	<ul style="list-style-type: none"> The first argument should be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported.
inf	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
insertMarker	Computer Vision System Toolbox	<ul style="list-style-type: none"> 'Color' input cannot be a cell array. position input cannot be a <code>cornerPoints</code> object. marker input must be constant. marker input cannot be 's'.
insertShape	Computer Vision System Toolbox	<ul style="list-style-type: none"> position input cannot be a cell array. 'Color' input cannot be a cell array. shape input must be constant.
int8, int16, int32	MATLAB	—

Function	Product	Remarks/Limitations
int8, int16, int32	Fixed-Point Designer	—
integralImage	Computer Vision System Toolbox	—
interp1	MATLAB	<ul style="list-style-type: none"> • Supports only linear and nearest interpolation methods. • Does not handle evenly spaced X indices separately. • X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.
interp2	MATLAB	<ul style="list-style-type: none"> • Supports only $5 \leq \text{nargin} \leq 7$. • XI and YI must be the same size. • Supports only 'linear' and 'nearest' methods. • For best results, supply X and Y as vectors. • When the X or Y inputs are not vectors, interp2 references only the first row of X and first column of Y. Supports "plaid" input for X and Y but does not verify that the input data is "plaid". • X and Y must contain monotonically increasing values. If your application provides monotonically decreasing values, first use flipplr and flipud to change X, Y, and Z to monotonically increasing form before calling interp2.

Function	Product	Remarks/Limitations
intersect	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example, zeros(1,0), to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double.

Function	Product	Remarks/Limitations
intfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
intmax	MATLAB	—
intmin	MATLAB	—
inv	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	MATLAB	—
ipermute	MATLAB	—
iptcheckconn	Image Processing Toolbox	All input arguments must be compile-time constants.
isa	MATLAB	—
iscell	MATLAB	—
ischar	MATLAB	—
iscolumn	MATLAB	—
iscolumn	Fixed-Point Designer	—
isdeployed	MATLAB Compiler	<ul style="list-style-type: none"> Returns true and false as appropriate for MEX and SIM targets Returns false for other targets

Function	Product	Remarks/Limitations
isempty	MATLAB	—
isempty	Fixed-Point Designer	—
isEpipoleInImage	Computer Vision System Toolbox	—
isequal	MATLAB	—
isequal	Fixed-Point Designer	—
isequaln	MATLAB	—
isfi	Fixed-Point Designer	—
isfield	MATLAB	<ul style="list-style-type: none"> Does not support cell input for second argument
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	MATLAB	—
isfinite	Fixed-Point Designer	—
isfloat	MATLAB	—
isinf	MATLAB	—
isinf	Fixed-Point Designer	—
isinteger	MATLAB	—
isletter	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127
islogical	MATLAB	—

Function	Product	Remarks/Limitations
ismac	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
ismatrix	MATLAB	—
ismcc	MATLAB Compiler	<ul style="list-style-type: none"> • Returns true and false as appropriate for MEX and SIM targets. • Returns false for other targets.
ismember	MATLAB	<ul style="list-style-type: none"> • The second input, B, must be sorted in ascending order. • Complex inputs must be single or double.
isnan	MATLAB	—
isnan	Fixed-Point Designer	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Designer	—
isnumericitype	Fixed-Point Designer	—
ispc	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.

Function	Product	Remarks/Limitations
isprime	MATLAB	<ul style="list-style-type: none"> For double precision input, the maximum value of A is $2^{32} - 1$. For single precision input, the maximum value of A is $2^{24} - 1$.
isreal	MATLAB	—
isreal	Fixed-Point Designer	—
isrow	MATLAB	—
isrow	Fixed-Point Designer	—
isscalar	MATLAB	—
isscalar	Fixed-Point Designer	—
issigned	Fixed-Point Designer	—
issorted	MATLAB	—
isspace	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127.
issparse	MATLAB	—
isstrprop	MATLAB	<ul style="list-style-type: none"> Supports only inputs from char and integer classes. Input values must be in the range 0-127.
isstruct	MATLAB	—
istrellis	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.

Function	Product	Remarks/Limitations
isunix	MATLAB	<ul style="list-style-type: none"> Returns true or false based on the MATLAB version used for code generation. Use only when the code generation target is S-function (Simulation) or MEX-function.
isvector	MATLAB	—
isvector	Fixed-Point Designer	—
kaiser	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
kaiserord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Computation performed at run time.
kron	MATLAB	—

Function	Product	Remarks/Limitations
label2rgb	Image Processing Toolbox	<p>Referring to the standard syntax: <code>RGB = label2rgb(L, map, zerocolor, order)</code></p> <ul style="list-style-type: none"> • Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. • <code>map</code> must be an <code>n-by-3</code>, double, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. • If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning. • If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.
lcm	MATLAB	—
ldivide	MATLAB	—
le	MATLAB	—
le	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
levinson	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
lineToBorderPoints	Computer Vision System Toolbox	—
linsolve	MATLAB	<ul style="list-style-type: none"> • The option structure must be a constant. • Supports only a scalar option structure input. It does not support arrays of option structures. • Only optimizes these cases: <ul style="list-style-type: none"> ▪ UT ▪ LT ▪ UHESS = true (the TRANS can be either true or false) ▪ SYM = true and POSDEF = true <p>Other options are equivalent to using <code>mldivide</code>.</p>
linspace	MATLAB	—

Function	Product	Remarks/Limitations
load	MATLAB	<ul style="list-style-type: none"> • Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use <code>coder.load</code>. • Does not support use of the function without assignment to a structure or array. For example, use <code>S = load(filename)</code>, not <code>load(filename)</code>. • The output <code>S</code> must be the name of a structure or array without any subscripting. For example, <code>S[i] = load('myFile.mat')</code> is not allowed. • Arguments to <code>load</code> must be compile-time constant strings. • Does not support loading objects. • If the MAT-file contains unsupported constructs, use <code>load(filename,variables)</code> to load only the supported constructs. • You cannot use <code>save</code> in a function intended for code generation. The code generation software does not support the <code>save</code> function. Furthermore, you cannot use <code>coder.extrinsic</code> with <code>save</code>. Prior to generating code, you can use <code>save</code> to save the workspace data to a MAT-file. <p>You must use <code>coder.varsizesize</code> to explicitly declare variable-size data loaded using the <code>load</code> function.</p>

Function	Product	Remarks/Limitations
log	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Designer	—
logspace	MATLAB	—
lower	MATLAB	<ul style="list-style-type: none"> Supports only char inputs. Input values must be in the range 0-127.
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lt	MATLAB	—
lt	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
lu	MATLAB	—
magic	MATLAB	—

Function	Product	Remarks/Limitations
matchFeatures	Computer Vision System Toolbox	—
max	MATLAB	—
max	Fixed-Point Designer	—
maxflat	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
mean	MATLAB	—
mean	Fixed-Point Designer	—
median	MATLAB	—
median	Fixed-Point Designer	—
meshgrid	MATLAB	—
min	MATLAB	—
min	Fixed-Point Designer	—
minus	MATLAB	—

Function	Product	Remarks/Limitations
minus	Fixed-Point Designer	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
mldivide	MATLAB	—
mod	MATLAB	<ul style="list-style-type: none"> Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors.
mode	MATLAB	<ul style="list-style-type: none"> Does not support third output argument <code>C</code> (cell array)
mpower	MATLAB	—
mpower	Fixed-Point Designer	<ul style="list-style-type: none"> The exponent input, k, must be constant; that is, its value must be known at compile time. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the

Function	Product	Remarks/Limitations
		ProductMode of the governing fimath.
mpy	Fixed-Point Designer	<ul style="list-style-type: none"> When you provide complex inputs to the mpy function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to On.
mrdivide	MATLAB	—
mrdivide	Fixed-Point Designer	—
mtimes	MATLAB	—
mtimes	Fixed-Point Designer	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both

Function	Product	Remarks/Limitations
		inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code> .
<code>NaN</code> or <code>nan</code>	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
<code>nargchk</code>	MATLAB	<ul style="list-style-type: none"> Output structure does not include stack information. <hr/> <p>Note <code>nargchk</code> will be removed in a future release.</p> <hr/>
<code>nargin</code>	MATLAB	—
<code>nargout</code>	MATLAB	<ul style="list-style-type: none"> For a function with no output arguments, returns 1 if called without a terminating semicolon. <hr/> <p>Note This behavior also affects extrinsic calls with no terminating semicolon. <code>nargout</code> is 1 for the called function in MATLAB.</p> <hr/>
<code>nargoutchk</code>	MATLAB	<ul style="list-style-type: none"> For code generation, you must supply <code>nargout</code> as the first input argument. Output structure does not include stack information.
<code>nchoosek</code>	MATLAB	<ul style="list-style-type: none"> When the first input is a scalar, it must be a non-negative integer.
<code>ndgrid</code>	MATLAB	—
<code>ndims</code>	MATLAB	—

Function	Product	Remarks/Limitations
ndims	Fixed-Point Designer	—
ne	MATLAB	—
ne	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
nearest	Fixed-Point Designer	—
nextpow2	MATLAB	—
nnz	MATLAB	—
nonzeros	MATLAB	—
norm	MATLAB	—
normest	MATLAB	—
not	MATLAB	—
nthroot	MATLAB	—
null	MATLAB	<ul style="list-style-type: none"> • Might return a different basis than MATLAB • Does not support rational basis option (second input)
num2hex	MATLAB	—
numberofelements	Fixed-Point Designer	<ul style="list-style-type: none"> • Returns the number of elements of <code>fi</code> objects in the generated code (works the same as <code>numel</code> for <code>fi</code> objects in generated code).
numel	MATLAB	<ul style="list-style-type: none"> • Returns the number of elements of <code>fi</code> objects in the generated code, rather than returning 1.

Function	Product	Remarks/Limitations
numerictype	Fixed-Point Designer	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>numerictype</code> object that is populated with the signal's data type and scaling information. Returns the data type when the input is a non-fixed-point signal. Use to create <code>numerictype</code> objects in the generated code.
nutallwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
ones	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
or	MATLAB	—
orth	MATLAB	<ul style="list-style-type: none"> Might return a different basis than MATLAB
padarray	Image Processing Toolbox	<p>Supports only up to 3-D inputs.</p> <p>Input arguments, <code>padval</code> and <code>direction</code>, are expected to be compile-time constants.</p>

Function	Product	Remarks/Limitations
parzenwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
pascal	MATLAB	—
permute	MATLAB	—
permute	Fixed-Point Designer	—
pi	MATLAB	—
pinv	MATLAB	—
planerot	MATLAB	—
plus	MATLAB	—
plus	Fixed-Point Designer	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
pol2cart	MATLAB	—
poly	MATLAB	<ul style="list-style-type: none"> Does not discard nonfinite input values Complex input produces complex output

Function	Product	Remarks/Limitations
poly2trellis	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
polyfit	MATLAB	—
polyval	MATLAB	—
pow2	Fixed-Point Designer	—
power	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when both X and Y are real, but <code>power(X,Y)</code> is complex. To get the complex result, make the input value X complex by passing in <code>complex(X)</code>. For example, <code>power(complex(X),Y)</code>. Generates an error during simulation and returns NaN in generated code when both X and Y are real, but <code>X.^Y</code> is complex. To get the complex result, make the input value X complex by using <code>complex(X)</code>. For example, <code>complex(X).^Y</code>.
power	Fixed-Point Designer	<ul style="list-style-type: none"> The exponent input, <i>k</i>, must be constant; that is, its value must be known at compile time.
primes	MATLAB	<ul style="list-style-type: none"> The largest supported double input is 2^{32}.
prod	MATLAB	—
qr	MATLAB	—

Function	Product	Remarks/Limitations
quad2d	MATLAB	<ul style="list-style-type: none"> Generates a warning if the size of the internal storage arrays is not large enough. If a warning occurs, a possible workaround is to divide the region of integration into pieces and sum the integrals over each piece.
quadgk	MATLAB	—
quantize	Fixed-Point Designer	—
quatconj	Aerospace Toolbox	—
quatdivide	Aerospace Toolbox	—
quatinv	Aerospace Toolbox	—
quatmod	Aerospace Toolbox	—
quatmultiply	Aerospace Toolbox	—
quatnorm	Aerospace Toolbox	—
quatnormalize	Aerospace Toolbox	—
rand	MATLAB	—
randi	MATLAB	—
randn	MATLAB	—
randperm	MATLAB	—
range	Fixed-Point Designer	—
rank	MATLAB	—

Function	Product	Remarks/Limitations
rcond	MATLAB	—
rcosfir	Communications System Toolbox	• Requires a Communications System Toolbox license to generate code.
rdivide	MATLAB	—
rdivide	Fixed-Point Designer	—
real	MATLAB	—
real	Fixed-Point Designer	—
reallog	MATLAB	—
realmax	MATLAB	—
realmax	Fixed-Point Designer	—
realmin	MATLAB	—
realmin	Fixed-Point Designer	—
realpow	MATLAB	—
realsqrt	MATLAB	—

Function	Product	Remarks/Limitations
rectwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
reinterprecast	Fixed-Point Designer	—
rem	MATLAB	<ul style="list-style-type: none"> Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors.
removefimath	Fixed-Point Designer	—
repmat	MATLAB	—
repmat	Fixed-Point Designer	—
resample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>.</p>

Function	Product	Remarks/Limitations
		For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.
rescale	Fixed-Point Designer	—
reshape	MATLAB	—
reshape	Fixed-Point Designer	—
rng	MATLAB	<ul style="list-style-type: none"> • For library and executable code generation targets, and for MEX targets when extrinsic calls are disabled, supports only the 'default' input and these generator inputs: <ul style="list-style-type: none"> ▪ 'twister' ▪ 'v4' ▪ 'v5normal' <p>For these targets, the output of <code>s=rng</code> in the generated code differs from the MATLAB output. You cannot return the output of <code>s=rng</code> from the generated code and pass it to <code>rng</code> in MATLAB.</p> <ul style="list-style-type: none"> • For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by <code>rng</code>.

Function	Product	Remarks/Limitations
roots	MATLAB	<ul style="list-style-type: none"> • Output is variable size • Output is complex • Roots may not be in the same order as MATLAB • Roots of poorly conditioned polynomials may not match MATLAB
rosser	MATLAB	—
rot90	MATLAB	—
round	MATLAB	—
round	Fixed-Point Designer	—
rsf2csf	MATLAB	—
schur	MATLAB	Might sometimes return a different Schur decomposition in generated code than in MATLAB.
sec	MATLAB	—
secd	MATLAB	—
sech	MATLAB	—

Function	Product	Remarks/Limitations
setdiff	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ Do not use [] to represent the empty set. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' and 'rows' options, the output <code>ia</code> is a column vector. If <code>ia</code> is empty, it is 0-by-1, never 0-by-0, even if the output <code>C</code> is 0-by-0. • When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>.
setfimath	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
setxor	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors with the same orientation. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example , zeros(1,0), to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' flag, the inputs must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double.
sfi	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
sgolay	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
shiftdim	MATLAB	Second argument must be a constant.
sign	MATLAB	—
sign	Fixed-Point Designer	—
sin	MATLAB	—
sin	Fixed-Point Designer	—
sind	MATLAB	—
single	MATLAB	—
single	Fixed-Point Designer	—
sinh	MATLAB	—
size	MATLAB	—
size	Fixed-Point Designer	—
sort	MATLAB	—
sort	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
sortrows	MATLAB	—
sosfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Computation performed at run time.
sph2cart	MATLAB	—
squeeze	MATLAB	—
sqrt	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
sqrt	Fixed-Point Designer	<ul style="list-style-type: none"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
sqrtm	MATLAB	—
std	MATLAB	—
storedInteger	Fixed-Point Designer	—
storedIntegerToDouble	Fixed-Point Designer	—
str2func	MATLAB	<ul style="list-style-type: none"> String must be constant/known at compile time
strcmp	MATLAB	<ul style="list-style-type: none"> Arguments must be computable at compile time.
strcmpi	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127.

Function	Product	Remarks/Limitations
strfind	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays. If <code>pattern</code> does not exist in <code>str</code>, returns <code>zeros(1,0)</code> not <code>[]</code>. To check for an empty return, use <code>isempty</code>. Inputs must be character row vectors.
strjust	MATLAB	—
strncmp	MATLAB	—
strncmpi	MATLAB	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0-127.
strep	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays. If <code>oldSubstr</code> does not exist in <code>origStr</code>, returns <code>blanks(0)</code>. To check for an empty return, use <code>isempty</code>. Inputs must be character row vectors.
strtok	MATLAB	—
strtrim	MATLAB	<ul style="list-style-type: none"> Supports only inputs from the <code>char</code> class. Input values must be in the range 0-127.
struct	MATLAB	—
structfun	MATLAB	<ul style="list-style-type: none"> Does not support the <code>ErrorHandler</code> option. The number of outputs must be less than or equal to three.
sub	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
sub2ind	MATLAB	<ul style="list-style-type: none"> The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported.
subsasgn	Fixed-Point Designer	—
subspace	MATLAB	—
subref	Fixed-Point Designer	—
sum	MATLAB	—
sum	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
svd	MATLAB	Uses a different SVD implementation than MATLAB. As the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
swapbytes	MATLAB	Inheritance of the class of the input to swapbytes in a MATLAB Function block is supported only when the class of the input is double. For non-double inputs, the input port data types must be specified, not inherited.
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—

Function	Product	Remarks/Limitations
taylorwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs must be constant <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
times	MATLAB	—
times	Fixed-Point Designer	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. When you provide complex inputs to the <code>times</code> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code>.
toeplitz	MATLAB	—
trace	MATLAB	—
trapz	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
triang	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
tril	MATLAB	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
tril	Fixed-Point Designer	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
triu	MATLAB	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
triu	Fixed-Point Designer	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
true	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.

Function	Product	Remarks/Limitations
tukeywin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
typecast	MATLAB	<ul style="list-style-type: none"> • Value of string input argument <code>type</code> must be lower case • You might receive a size error when you use <code>typecast</code> with inheritance of input port data types in MATLAB Function blocks. To avoid this error, specify the block’s input port data types explicitly.
ufi	Fixed-Point Designer	—
uint8, uint16, uint32	MATLAB	—
uint8, uint16, uint32	Fixed-Point Designer	—
uminus	MATLAB	—
uminus	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
union	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors with the same orientation. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example , zeros(1,0), to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double.

Function	Product	Remarks/Limitations
unique	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ The input A must be a vector. If you specify the 'legacy' option, the input A must be a row vector. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example , <code>zeros(1,0)</code>, to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'rows' option and the 'legacy' option, outputs ia and ic are column vectors. If these outputs are empty, they are 0-by-1, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' option, the input A must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double.

Function	Product	Remarks/Limitations
unwrap	MATLAB	<ul style="list-style-type: none"> Row vector input is only supported when the first two inputs are vectors and nonscalar Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors
upfirdn	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
uplus	MATLAB	—
uplus	Fixed-Point Designer	—
upper	MATLAB	<ul style="list-style-type: none"> Supports only char inputs. Input values must be in the range 0-127.
upperbound	Fixed-Point Designer	—

Function	Product	Remarks/Limitations
upsample	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Either declare input <code>n</code> as constant, or use the <code>assert</code> function in the calling function to set upper bounds for <code>n</code>. For example, <code>assert(n<10)</code>
vander	MATLAB	—
var	MATLAB	—
vertcat	Fixed-Point Designer	—
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Does not support the case where <code>A</code> is a matrix • Does not support partial (abbreviated) strings of <code>biased</code>, <code>unbiased</code>, <code>coeff</code>, or <code>none</code> • Computation performed at run time.
xor	MATLAB	—

Function	Product	Remarks/Limitations
yulewalk	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 19-46.</p>
zeros	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
zp2tf	MATLAB	—

Functions Supported for Code Generation — Categorical List

In this section...

- “Aerospace Toolbox Functions” on page 4-80
- “Arithmetic Operator Functions” on page 4-81
- “Bit-Wise Operation Functions” on page 4-81
- “Casting Functions” on page 4-82
- “Communications System Toolbox Functions” on page 4-82
- “Complex Number Functions” on page 4-82
- “Computer Vision System Toolbox Functions” on page 4-83
- “Data and File Management Functions” on page 4-84
- “Data Type Functions” on page 4-85
- “Derivative and Integral Functions” on page 4-85
- “Discrete Math Functions” on page 4-85
- “Error Handling Functions” on page 4-86
- “Exponential Functions” on page 4-86
- “Filtering and Convolution Functions” on page 4-87
- “Fixed-Point Designer Functions” on page 4-87
- “Histogram Functions” on page 4-96
- “Image Processing Toolbox Functions” on page 4-96
- “Input and Output Functions” on page 4-98
- “Interpolation and Computational Geometry Functions” on page 4-99
- “Linear Algebra” on page 4-99
- “Logical Operator Functions” on page 4-99
- “MATLAB® Compiler™ Functions” on page 4-100
- “MATLAB Desktop Environment Functions” on page 4-100
- “Matrix and Array Functions” on page 4-100

In this section...
“Nonlinear Numerical Methods” on page 4-104
“Polynomial Functions” on page 4-105
“Relational Operator Functions” on page 4-105
“Rounding and Remainder Functions” on page 4-105
“Set Functions” on page 4-106
“Signal Processing Functions in MATLAB” on page 4-106
“Signal Processing Toolbox Functions” on page 4-107
“Special Values” on page 4-111
“Specialized Math” on page 4-112
“Statistical Functions” on page 4-112
“String Functions” on page 4-113
“Structure Functions” on page 4-114
“Trigonometric Functions” on page 4-114

Aerospace Toolbox Functions

Function	Description
quatconj	Calculate conjugate of quaternion
quatdivide	Divide quaternion by another quaternion
quatinv	Calculate inverse of quaternion
quatmod	Calculate modulus of quaternion
quatmultiply	Calculate product of two quaternions
quatnorm	Calculate norm of quaternion
quatnormalize	Normalize quaternion

Arithmetic Operator Functions

See Arithmetic Operators for detailed descriptions of the following operator equivalent functions.

Function	Description
ctranspose	Complex conjugate transpose (')
idivide	Integer division with rounding option
isa	Determine if input is object of given class
ldivide	Left array divide
minus	Minus (-)
mldivide	Left matrix divide (\)
mpower	Equivalent of array power operator (.^)
mrdivide	Right matrix divide
mtimes	Matrix multiply (*)
plus	Plus (+)
power	Array power
rdivide	Right array divide
times	Array multiply
transpose	Matrix transpose (')
uminus	Unary minus (-)
uplus	Unary plus (+)

Bit-Wise Operation Functions

Function	Description
flintmax	Largest consecutive integer in floating-point format
swapbytes	Swap byte ordering

Casting Functions

Data Type	Description
cast	Cast variable to different data type
char	Create character array (string)
class	Query class of object argument
double	Convert to double-precision floating point
int8, int16, int32	Convert to signed integer data type
logical	Convert to Boolean true or false data type
single	Convert to single-precision floating point
typecast	Convert data types without changing underlying data
uint8, uint16, uint32	Convert to unsigned integer data type

Communications System Toolbox Functions

Function	Remarks/Limitations
bi2de	—
de2bi	—
istrellis	—
poly2trellis	—
rcosfir	—

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components
conj	Return the conjugate of a complex number
imag	Return the imaginary part of a complex number
isnumeric	Return true for numeric arrays

Function	Description
isreal	Return false (0) for a complex number
isscalar	Return true if array is a scalar
real	Return the real part of a complex number
unwrap	Correct phase angles to produce smoother phase plots

Computer Vision System Toolbox Functions

Function	Description
binaryFeatures	Object for storing binary feature vectors
epipolarLine	Compute epipolar lines for stereo images
estimateFundamentalMatrix	Estimate fundamental matrix from corresponding points in stereo image
estimateUncalibratedRectification	Uncalibrated stereo rectification
extractFeatures	Extract interest point descriptors
insertMarker	Insert markers in image or video

Function	Description
<code>insertShape</code>	Insert shapes in image or video
<code>integralImage</code>	Compute integral image
<code>isEpipoleInImage</code>	Determine whether image contains epipole
<code>vision.KalmanFilter</code>	Kalman filter for object tracking
<code>lineToBorderPoints</code>	Intersection points of lines in image and image border
<code>matchFeatures</code>	Find matching image features

Data and File Management Functions

Function	Description
<code>computer</code>	Information about computer on which MATLAB software is running
<code>fclose</code>	Close one or all open files
<code>fopen</code>	Open file, or obtain information about open files
<code>fprintf</code>	Write data to text file
<code>load</code>	Load data from MAT-file into workspace

Data Type Functions

Function	Description
deal	Distribute inputs to outputs
iscell	Determine whether input is cell array
nargchk	Validate number of input arguments Note nargchk will be removed in a future release.
nargoutchk	Validate number of output arguments
str2func	Construct function handle from function name string
structfun	Apply function to each field of scalar structure

Derivative and Integral Functions

Function	Description
cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives
gradient	Numerical gradient
trapz	Trapezoidal numerical integration

Discrete Math Functions

Function	Description
factor	Return a row vector containing the prime factors of n
gcd	Return an array containing the greatest common divisors of the corresponding elements of integer arrays
isprime	Array elements that are prime numbers

Function	Description
lcm	Least common multiple of corresponding elements in arrays
nchoosek	Binomial coefficient or all combinations
primes	Generate list of prime numbers

Error Handling Functions

Function	Description
assert	Generate error when condition is violated
error	Display message and abort function

Exponential Functions

Function	Description
exp	Exponential
expm	Matrix exponential
expm1	Compute $\exp(x) - 1$ accurately for small values of x
factorial	Factorial function
log	Natural logarithm
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
log10	Common (base 10) logarithm
log1p	Compute $\log(1+x)$ accurately for small values of x
nextpow2	Next higher power of 2
nthroot	Real n th root of real numbers
reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root

Filtering and Convolution Functions

Function	Description
<code>conv</code>	Convolution and polynomial multiplication
<code>conv2</code>	2-D convolution
<code>convn</code>	N-D convolution
<code>deconv</code>	Deconvolution and polynomial division
<code>detrend</code>	Remove linear trends
<code>filter</code>	1-D digital filter
<code>filter2</code>	2-D digital filter

Fixed-Point Designer Functions

In addition to function-specific limitations listed in the table, the following general limitations apply to the use of Fixed-Point Designer functions in generated code or with `fiaccel`:

- `fipref` and quantizer objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fi`math or `numericType` of a given `fi` variable after that variable has been created.
- The boolean value of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- You can use parallel for (`parfor`) loops in code compiled with `fiaccel`, but those loops are treated like regular for loops.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.

- The general limitations of C/C++ code generated from MATLAB apply. For more information, see “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-12.

Function	Remarks/Limitations
abs	N/A
accumneg	N/A
accumpos	N/A
add	N/A
all	N/A
any	N/A
atan2	N/A
bitand	Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	N/A
bitcmp	N/A
bitconcat	N/A
bitget	N/A
bitor	Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	N/A
bitreplicate	N/A
bitrol	N/A
bitror	N/A
bitset	N/A
bitshift	N/A
bitsliceget	N/A
bitsll	N/A
bitsra	N/A
bitsrl	N/A
bitxor	Not supported for slope-bias scaled <code>fi</code> objects.

Function	Remarks/Limitations
bitxorreduce	N/A
ceil	N/A
complex	N/A
conj	N/A
conv	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. • For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
convergent	N/A
cordicabs	Variable-size signals are not supported.
cordicangle	Variable-size signals are not supported.
cordicatan2	Variable-size signals are not supported.
cordiccart2pol	Variable-size signals are not supported.
cordiccxp	Variable-size signals are not supported.
cordiccos	Variable-size signals are not supported.
cordicpol2cart	Variable-size signals are not supported.
cordicrotate	Variable-size signals are not supported.
cordicsin	Variable-size signals are not supported.
cordicsincos	Variable-size signals are not supported.
cos	N/A
ctranspose	N/A

Function	Remarks/Limitations
diag	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.
divide	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Complex and imaginary divisors are not supported. Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
double	N/A
end	N/A
eps	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	Not supported for fixed-point signals with different biases.
fi	<ul style="list-style-type: none"> The default constructor syntax without any input arguments is not supported. If the <code>numericType</code> is not fully specified, the input to <code>fi</code> must be a constant, a <code>fi</code>, a <code>single</code>, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows <code>fi</code> to autoscale its fraction length based on the known data type of the input. <code>numericType</code> object information must be available for nonfixed-point Simulink inputs.
filter	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
fimath	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>fimath</code> object. You define this object in the MATLAB Function block dialog in the Model Explorer. Use to create <code>fimath</code> objects in the generated code.
fix	N/A

Function	Remarks/Limitations
<code>fixed.Quantizer</code>	N/A
<code>floor</code>	N/A
<code>ge</code>	Not supported for fixed-point signals with different biases.
<code>get</code>	The syntax <code>structure = get(o)</code> is not supported.
<code>getlsb</code>	N/A
<code>getmsb</code>	N/A
<code>gt</code>	Not supported for fixed-point signals with different biases.
<code>hdlram</code>	N/A
<code>horzcat</code>	N/A
<code>imag</code>	N/A
<code>int8, int16, int32</code>	N/A
<code>iscolumn</code>	N/A
<code>isempty</code>	N/A
<code>isequal</code>	N/A
<code>isfi</code>	N/A
<code>isfimath</code>	N/A
<code>isfimathlocal</code>	N/A
<code>isfinite</code>	N/A
<code>isinf</code>	N/A
<code>isnan</code>	N/A
<code>isnumeric</code>	N/A
<code>isnumerictype</code>	N/A
<code>isreal</code>	N/A
<code>isrow</code>	N/A
<code>isscalar</code>	N/A
<code>issigned</code>	N/A

Function	Remarks/Limitations
isvector	N/A
le	Not supported for fixed-point signals with different biases.
length	N/A
logical	N/A
lowerbound	N/A
lsb	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lt	Not supported for fixed-point signals with different biases.
max	N/A
mean	N/A
median	N/A
min	N/A
minus	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
mpower	<ul style="list-style-type: none"> When the exponent <code>k</code> is a variable and the input is a scalar, the <code>ProductMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code>. When the exponent <code>k</code> is a variable and the input is not scalar, the <code>SumMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code>. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>SpecifyPrecision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when

Function	Remarks/Limitations
	the first input, <i>a</i> , is nonscalar. However, when <i>a</i> is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code> .
<code>mpy</code>	When you provide complex inputs to the <code>mpy</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code> .
<code>mrdivide</code>	N/A
<code>mtimes</code>	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>SpecifyPrecision</code> or <code>KeepLSB</code>. • For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
<code>ndims</code>	N/A
<code>ne</code>	Not supported for fixed-point signals with different biases.
<code>nearest</code>	N/A
<code>numberofelements</code>	<code>numberofelements</code> and <code>numel</code> both work the same as MATLAB <code>numel</code> for <code>fi</code> objects in the generated code.

Function	Remarks/Limitations
<code>numericType</code>	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>numericType</code> object that is populated with the signal's data type and scaling information. Returns the data type when the input is a nonfixed-point signal. Use to create <code>numericType</code> objects in generated code.
<code>permute</code>	N/A
<code>plus</code>	Any non- <code>fi</code> inputs must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
<code>pow2</code>	N/A
<code>power</code>	When the exponent <code>k</code> is a variable, the <code>ProductMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code> .
<code>quantize</code>	N/A
<code>range</code>	N/A
<code>rdivide</code>	N/A
<code>real</code>	N/A
<code>realmax</code>	N/A
<code>realmin</code>	N/A
<code>reinterpretcast</code>	N/A
<code>removefimath</code>	N/A
<code>repmat</code>	N/A
<code>rescale</code>	N/A
<code>reshape</code>	N/A
<code>round</code>	N/A
<code>setfimath</code>	N/A
<code>sfi</code>	N/A
<code>sign</code>	N/A
<code>sin</code>	N/A

Function	Remarks/Limitations
single	N/A
size	N/A
sort	N/A
sqrt	<ul style="list-style-type: none"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
times	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. When you provide complex inputs to the times function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to On.
transpose	N/A
tril	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
triu	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
ufi	N/A
uint8, uint16, uint32	N/A
uminus	N/A
uplus	N/A

Function	Remarks/Limitations
upperbound	N/A
vertcat	N/A

Histogram Functions

Function	Description
hist	Non-graphical histogram
histc	Histogram count

Image Processing Toolbox Functions

You must have the MATLAB Coder and Image Processing Toolbox software installed to generate C/C++ code from MATLAB for these functions.

Function	Remarks/Limitations
bwlookup	For best results, specify an input image of class <code>logical</code> .
bwmorph	The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code> .
conndef	All input arguments must be compile-time constants.
fspecial	All inputs must be compile-time constants. Expressions or variables are allowed if their values do not change.
imcomplement	Does not support <code>int64</code> and <code>uint64</code> data types.

Function	Remarks/Limitations
imfill	<p>The optional input connectivity, <code>conn</code> and the string 'holes' must be a compile time constants.</p> <p>Supports only up to 3-D inputs. (No N-D support.)</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code> is not supported in code generation.</p> <p><code>locations</code> can be a P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-by-<code>ndims(I)</code> matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile-time, you cannot change it at run-time. However, the number of points in <code>locations</code> can be varied at run-time.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmax	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmin	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imreconstruct	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imregionalmax	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Function	Remarks/Limitations
<code>imregionalmin</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
<code>iptcheckconn</code>	All input arguments must be compile-time constants.
<code>label2rgb</code>	<p>Referring to the standard syntax:</p> <pre>RGB = label2rgb(L, map, zerocolor, order)</pre> <ul style="list-style-type: none"> • Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. • <code>map</code> must be an <code>n-by-3, double</code>, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. • If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning. • If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.
<code>padarray</code>	<p>Support only up to 3-D inputs.</p> <p>Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.</p>

Input and Output Functions

Function	Description
<code>nargin</code>	Return the number of input arguments a user has supplied
<code>nargout</code>	Return the number of output return values a user has requested

Interpolation and Computational Geometry Functions

Function	Description
cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
meshgrid	Generate X and Y arrays for 3-D plots
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Linear Algebra

Function	Description
linsolve	Solve linear system of equations
null	Null space
orth	Range space of matrix
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root

Logical Operator Functions

Function	Description
and	Logical AND (&&)
bitand	Bitwise AND
bitcmp	Bitwise complement

Function	Description
bitget	Bit at specified position
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
not	Logical NOT (~)
or	Logical OR ()
xor	Logical exclusive-OR

MATLAB Compiler Functions

Function	Description
isdeployed	Determine whether code is running in deployed or MATLAB mode
ismcc	Test if code is running during compilation process (using mcc)

MATLAB Desktop Environment Functions

Function	Description
ismac	Determine if version is for Mac OS X platform
ispc	Determine if version is for Windows (PC) platform
isunix	Determine if version is for UNIX® platform

Matrix and Array Functions

Function	Description
abs	Return absolute value and complex magnitude of an array
all	Test if all elements are nonzero

Function	Description
angle	Phase angle
any	Test for any nonzero elements
blkdiag	Construct block diagonal matrix from input arguments
bsxfun	Applies element-by-element binary operation to two arrays with singleton expansion enabled
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly
compan	Companion matrix
cond	Condition number of a matrix with respect to inversion
cov	Covariance matrix
cross	Vector cross product
cumprod	Cumulative product of array elements
cumsum	Cumulative sum of array elements
det	Matrix determinant
diag	Return a matrix formed around the specified diagonal vector and the specified diagonal (0, 1, 2,...) it occupies
diff	Differences and approximate derivatives
dot	Vector dot product
eig	Eigenvalues and eigenvectors
eye	Identity matrix
false	Return an array of 0s for the specified dimensions
find	Find indices and values of nonzero elements
flipdim	Flip array along specified dimension
flipplr	Flip matrix left to right
flipud	Flip matrix up to down
full	Convert sparse matrix to full matrix
hadamard	Hadamard matrix

Function	Description
hankel	Hankel matrix
hilb	Hilbert matrix
ind2sub	Subscripts from linear index
inv	Inverse of a square matrix
invhilb	Inverse of Hilbert matrix
ipermute	Inverse permute dimensions of array
iscolumn	True if input is a column vector
isempty	Determine whether array is empty
isequal	Test arrays for equality
isequaln	Test arrays for equality, treating NaNs as equal
isfinite	Detect finite elements of an array
isfloat	Determine if input is floating-point array
isinf	Detect infinite elements of an array
isinteger	Determine if input is integer array
islogical	Determine if input is logical array
ismatrix	True if input is a matrix
isnan	Detect NaN elements of an array
isrow	True if input is a row vector
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
kron	Kronecker tensor product
length	Return the length of a matrix
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
lu	Matrix factorization
magic	Magic square

Function	Description
max	Maximum elements of a matrix
min	Minimum elements of a matrix
ndgrid	Generate arrays for N-D functions and interpolation
ndims	Number of dimensions
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
norm	Vector and matrix norms
normest	2-norm estimate
numel	Number of elements in array or subscripted array
ones	Create a matrix of all 1s
pascal	Pascal matrix
permute	Rearrange dimensions of array
pinv	Pseudoinverse of a matrix
planerot	Givens plane rotation
prod	Product of array element
qr	Orthogonal-triangular decomposition
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed random numbers
randperm	Random permutation
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
repmat	Replicate and tile an array
reshape	Reshape one array into the dimensions of another
rng	Control random number generation
rosser	Classic symmetric eigenvalue test problem

Function	Description
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sign	Signum function
size	Return the size of a matrix
sort	Sort elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
sub2ind	Single index from subscripts
subspace	Angle between two subspaces
sum	Sum of matrix elements
toeplitz	Toeplitz matrix
trace	Sum of diagonal elements
tril	Extract lower triangular part
triu	Extract upper triangular part
true	Return an array of logical (Boolean) 1s for the specified dimensions
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix
zeros	Create a matrix of all zeros

Nonlinear Numerical Methods

Function	Description
fzero	Find root of continuous function of one variable
quad2d	Numerically evaluate double integral over planar region
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Polynomial Functions

Function	Description
poly	Polynomial with specified roots
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation
roots	Polynomial roots

Relational Operator Functions

Function	Description
eq	Equal (==)
ge	Greater than or equal to (>=)
gt	Greater than (>)
le	Less than or equal to (<=)
lt	Less than (<)
ne	Not equal (~=)

Rounding and Remainder Functions

Function	Description
ceil	Round toward plus infinity
ceil	Round toward positive infinity
convergent	Round toward nearest integer with ties rounding to nearest even integer
fix	Round toward zero
fix	Round toward zero
floor	Round toward minus infinity
floor	Round toward negative infinity
mod	Modulus (signed remainder after division)

Function	Description
nearest	Round toward nearest integer with ties rounding toward positive infinity
rem	Remainder after division
round	Round toward nearest integer
round	Round fi object toward nearest integer or round input data using quantizer object

Set Functions

Function	Description
intersect	Set intersection of two arrays
ismember	Array elements that are members of set array
issorted	Determine whether set elements are in sorted order
setdiff	Set difference of two arrays
setxor	Set exclusive OR of two arrays
union	Set union of two arrays
unique	Unique values in array

Signal Processing Functions in MATLAB

Function	Description
chol	Cholesky factorization
conv	Convolution and polynomial multiplication
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
filter	Filter a data sequence using a digital filter that works for both real and complex inputs

Function	Description
freqspace	Frequency spacing for frequency response
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse discrete Fourier transform shift
svd	Singular value decomposition
zp2tf	Convert zero-pole-gain filter parameters to transfer function form

Signal Processing Toolbox Functions

These functions do not support variable-size inputs, you must define the size and type of the function inputs. For more information, see “Specifying Inputs in Code Generation from MATLAB”.

Note Many Signal Processing Toolbox functions require constant inputs in generated code. To specify a constant input for `codegen`, use `coder.Constant`.

Function	Remarks/Limitations
barthannwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bartlett	Window length must be a constant. Expressions or variables are allowed if their values do not change.
besselap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
bitrevorder	—
blackman	Window length must be a constant. Expressions or variables are allowed if their values do not change.
blackmanharris	Window length must be a constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
bohmanwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
buttap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
butter	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
buttord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cfirpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
dct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
downsample	—
dpss	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
ellip	Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
filtfilt	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
fir1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fir2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firrcos	All inputs must be constants. Expressions or variables are allowed if their values do not change.
flattopwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
freqz	freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See “freqz With No Output Arguments”.
gaussfir	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
gausswin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hamming	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hann	All inputs must be constant. Expressions or variables are allowed if their values do not change.
idct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
intfilt	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiser	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiserord	—
levinson	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.
maxflat	All inputs must be constant. Expressions or variables are allowed if their values do not change.
nutallwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
parzenwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
rectwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
resample	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
sgolay	All inputs must be constant. Expressions or variables are allowed if their values do not change.
sosfilt	—

Function	Remarks/Limitations
taylorwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
triang	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tukeywin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
upfirdn	<ul style="list-style-type: none"> Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. Variable-size inputs are not supported.
upsample	Either declare input <code>n</code> as constant, or use the <code>assert</code> function in the calling function to set upper bounds for <code>n</code> . For example, <pre>assert(n<10)</pre>
xcorr	—
yulewalk	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Special Values

Symbol	Description
eps	Floating-point relative accuracy
inf	IEEE® arithmetic representation for positive infinity
intmax	Largest possible value of specified integer type
intmin	Smallest possible value of specified integer type
NaN or nan	Not a number
pi	Ratio of the circumference to the diameter for a circle
realmax	Largest positive floating-point number
realmin	Smallest positive floating-point number

Specialized Math

Symbol	Description
beta	Beta function
betainc	Incomplete beta function
betaln	Logarithm of beta function
ellipke	Complete elliptic integrals of first and second kind
erf	Error function
erfc	Complementary error function
erfcinv	Inverse of complementary error function
erfcx	Scaled complementary error function
erfinv	Inverse error function
expint	Exponential integral
gamma	Gamma function
gammainc	Incomplete gamma function
gammaln	Logarithm of the gamma function

Statistical Functions

Function	Description
corrcoef	Correlation coefficients
mean	Average or mean value of array
median	Median value of array
mode	Most frequent values in array
std	Standard deviation
var	Variance

String Functions

Function	Description
bin2dec	Convert binary number string to decimal number
bitmax	Maximum double-precision floating-point integer
blanks	Create string of blank characters
char	Create character array (string)
deblank	Strip trailing blanks from end of string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
ischar	True for character array (string)
isletter	Array elements that are alphabetic letters
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
lower	Convert string to lowercase
num2hex	Convert singles and doubles to IEEE hexadecimal strings
strcmp	Compare strings (case sensitive)
strcmpi	Compare strings (case insensitive)
strfind	Find one string within another
strjust	Justify character array
strncmp	Compare first n characters of strings (case sensitive)
strncmpi	Compare first n characters of strings (case insensitive)
strrep	Find and replace substring
strtok	Selected parts of string
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

Structure Functions

Function	Description
isfield	Determine whether input is structure array field
struct	Create structure
isstruct	Determine whether input is a structure

Trigonometric Functions

Function	Description
acos	Inverse cosine
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse cosecant and inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atan2	Four quadrant inverse tangent
atan2d	Four-quadrant inverse tangent, result in degrees
atand	Inverse tangent; result in degrees

Function	Description
atanh	Inverse hyperbolic tangent
cos	Cosine
cosd	Cosine; result in degrees
cosh	Hyperbolic cosine
cot	Cotangent; result in radians
cotd	Cotangent; result in degrees
coth	Hyperbolic cotangent
csc	Cosecant; result in radians
cscd	Cosecant; result in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant; result in radians
secd	Secant; result in degrees
sech	Hyperbolic secant
sin	Sine
sind	Sine; result in degrees
sinh	Hyperbolic sine
tan	Tangent
tand	Tangent; result in degrees
tanh	Hyperbolic tangent

Defining MATLAB Variables for C/C++ Code Generation

- “Variables Definition for Code Generation” on page 5-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 5-3
- “Eliminate Redundant Copies of Variables in Generated Code” on page 5-7
- “Reassignment of Variable Properties” on page 5-9
- “Define and Initialize Persistent Variables” on page 5-10
- “Reuse the Same Variable with Different Properties” on page 5-11
- “Avoid Overflows in for-Loops” on page 5-16
- “Supported Variable Types” on page 5-18

Variables Definition for Code Generation

In the MATLAB language, variables can change their properties dynamically at run time so you can use the same variable to hold a value of any class, size, or complexity. For example, the following code works in MATLAB:

```
function x = foo(c) %#codegen
if(c>0)
    x = 0;
else
    x = [1 2 3];
end
disp(x);
end
```

However, statically-typed languages like C must be able to determine variable properties at compile time. Therefore, for C/C++ code generation, you must explicitly define the class, size, and complexity of variables in MATLAB source code before using them. For example, rewrite the above source code with a definition for *x*:

```
function x = foo(c) %#codegen
x = zeros(1,3);
if(c>0)
    x = 0;
else
    x = [1 2 3];
end
disp(x);
end
```

For more information, see “Best Practices for Defining Variables for C/C++ Code Generation” on page 5-3.

Best Practices for Defining Variables for C/C++ Code Generation

In this section...

“Define Variables By Assignment Before Using Them” on page 5-3

“Use Caution When Reassigning Variables” on page 5-6

“Use Type Cast Operators in Variable Definitions” on page 5-6

“Define Matrices Before Assigning Indexed Variables” on page 5-6

Define Variables By Assignment Before Using Them

For C/C++ code generation, you should explicitly and unambiguously define the class, size, and complexity of variables before using them in operations or returning them as outputs. Define variables by assignment, but note that the assignment copies not only the value, but also the size, class, and complexity represented by that value to the new variable. For example:

Assignment:	Defines:
<code>a = 14.7;</code>	a as a real double scalar.
<code>b = a;</code>	b with properties of a (real double scalar).
<code>c = zeros(5,2);</code>	c as a real 5-by-2 array of doubles.
<code>d = [1 2 3 4 5; 6 7 8 9 0];</code>	d as a real 5-by-2 array of doubles.
<code>y = int16(3);</code>	y as a real 16-bit integer scalar.

Define properties this way so that the variable is defined on the required execution paths during C/C++ code generation (see Defining a Variable for Multiple Execution Paths on page 5-4).

The data that you assign to a variable can be a scalar, matrix, or structure. If your variable is a structure, define the properties of each field explicitly (see Defining Fields in a Structure on page 5-5).

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in “Eliminate Redundant Copies of Variables in Generated Code” on page 5-7.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see “Define and Initialize Persistent Variables” on page 5-10.

Defining a Variable for Multiple Execution Paths

Consider the following MATLAB code:

```
...
if c > 0
    x = 11;
end
% Later in your code ...
if c > 0
    use(x);
end
...
```

Here, x is assigned only if $c > 0$ and used only when $c > 0$. This code works in MATLAB, but generates a compilation error during code generation because it detects that x is undefined on some execution paths (when $c \leq 0$),.

To make this code suitable for code generation, define x before using it:

```
x = 0;
...
if c > 0
    x = 11;
end
% Later in your code ...
if c > 0
    use(x);
end
...
```

Defining Fields in a Structure

Consider the following MATLAB code:

```
...
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    s.b = 12;
end
% Try to use s
use(s);
...
```

Here, the first part of the `if` statement uses only the field `a`, and the `else` clause uses fields `a` and `b`. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see “Structure Definition for Code Generation” on page 8-2.

To make this code suitable for C/C++ code generation, define all fields of `s` before using it.

```
...
% Define all fields in structure s
s = struct( a ,0, b , 0);
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    s.b = 12;
end
% Use s
use(s);
...
```

Use Caution When Reassigning Variables

In general, you should adhere to the "one variable/one type" rule for C/C++ code generation; that is, each variable must have a specific class, size and complexity. Generally, if you reassign variable properties after the initial assignment, you get a compilation error during code generation, but there are exceptions, as described in “Reassignment of Variable Properties” on page 5-9.

Use Type Cast Operators in Variable Definitions

By default, constants are of type double. To define variables of other types, you can use type cast operators in variable definitions. For example, the following code defines variable `y` as an integer:

```
...
x = 15; % x is of type double by default.
y = uint8(x); % y has the value of x, but cast to uint8.
...
```

Define Matrices Before Assigning Indexed Variables

When generating C/C++ code from MATLAB, you cannot grow a variable by writing into an element beyond its current size. Such indexing operations produce run-time errors. You must define the matrix first before assigning values to its elements.

For example, the following initial assignment is not allowed for code generation:

```
g(3,2) = 14.6; % Not allowed for creating g
             % OK for assigning value once created
```

For more information about indexing matrices, see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-32.

Eliminate Redundant Copies of Variables in Generated Code

In this section...

“When Redundant Copies Occur” on page 5-7

“How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 5-7

“Defining Uninitialized Variables” on page 5-8

When Redundant Copies Occur

During C/C++ code generation, MATLAB checks for statements that attempt to access uninitialized memory. If it detects execution paths where a variable is used but is potentially not defined, it generates a compile-time error. To prevent these errors, define variables by assignment before using them in operations or returning them as function outputs.

Note, however, that variable assignments not only copy the properties of the assigned data to the new variable, but also initialize the new variable to the assigned value. This forced initialization sometimes results in redundant copies in C/C++ code. To eliminate redundant copies, define uninitialized variables by using the `coder.nullcopy` function, as described in “How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 5-7.

How to Eliminate Redundant Copies by Defining Uninitialized Variables

- 1 Define the variable with `coder.nullcopy`.
- 2 Initialize the variable before reading it.

When the uninitialized variable is an array, you must initialize all of its elements before passing the array as an input to a function or operator — even if the function or operator does not read from the uninitialized portion of the array.

What happens if you access uninitialized data?

Uninitialized memory contains arbitrary values. Therefore, accessing uninitialized data may lead to segmentation violations or nondeterministic program behavior (different runs of the same program may yield inconsistent results).

Defining Uninitialized Variables

In the following code, the assignment statement `X = zeros(1,N)` not only defines `X` to be a 1-by-5 vector of real doubles, but also initializes each element of `X` to zero.

```
function X = fcn %#codegen

N = 5;
X = zeros(1,N);
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    else
        X(i) = 0;
    end
end
```

This forced initialization creates an extra copy in the generated code. To eliminate this overhead, use `coder.nullcopy` in the definition of `X`:

```
function X = fcn2 %#codegen

N = 5;
X = coder.nullcopy(zeros(1,N));
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    else
        X(i) = 0;
    end
end
```

Reassignment of Variable Properties

For C/C++ code generation, there are certain variables that you can reassign after the initial assignment with a value of different class, size, or complexity:

Dynamically sized variables

A variable can hold values that have the same class and complexity but different sizes. If the size of the initial assignment is not constant, the variable is dynamically sized in generated code. For more information, see “Variable-Size Data”.

Variables reused in the code for different purposes

You can reassign the type (class, size, and complexity) of a variable after the initial assignment if each occurrence of the variable can have only one type. In this case, the variable is renamed in the generated code to create multiple independent variables. For more information, see “Reuse the Same Variable with Different Properties” on page 5-11.

Define and Initialize Persistent Variables

Persistent variables are local to the function in which they are defined, but they retain their values in memory between calls to the function. To define persistent variables for C/C++ code generation, use the `persistent` statement, as in this example:

```
persistent PROD_X;
```

The definition should appear at the top of the function body, after the header and comments, but before the first use of the variable. During code generation, the value of the persistent variable is initialized to an empty matrix by default. You can assign your own value after the definition by using the `isempty` statement, as in this example:

```
function findProduct(inputvalue) %#codegen
persistent PROD_X

if isempty(PROD_X)
    PROD_X = 1;
end
PROD_X = PROD_X * inputvalue;
end
```


Reuse the Same Variable with Different Properties

In this section...

“When You Can Reuse the Same Variable with Different Properties” on page 5-11

“When You Cannot Reuse Variables” on page 5-12

“Limitations of Variable Reuse” on page 5-14

When You Can Reuse the Same Variable with Different Properties

You can reuse (reassign) an input, output, or local variable with different class, size, or complexity if MATLAB can unambiguously determine the properties of each occurrence of this variable during C/C++ code generation. If so, MATLAB creates separate uniquely named local variables in the generated code. You can view these renamed variables in the code generation report (see “Viewing Variables in Your MATLAB Code” on page 19-186).

A common example of variable reuse is in `if-elseif-else` or `switch-case` statements. For example, the following function `example1` first uses the variable `t` in an `if` statement, where it holds a scalar double, then reuses `t` outside the `if` statement to hold a vector of doubles.

```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
```

To compile this example and see how MATLAB renames the reused variable `t`, see Variable Reuse in an `if` Statement on page 5-12.

When You Cannot Reuse Variables

You cannot reuse (reassign) variables if it is not possible to determine the class, size, and complexity of an occurrence of a variable unambiguously during code generation. In this case, variables cannot be renamed and a compilation error occurs.

For example, the following `example2` function assigns a fixed-point value to `x` in the `if` statement and reuses `x` to store a matrix of doubles in the `else` clause. It then uses `x` after the `if-else` statement. This function generates a compilation error because after the `if-else` statement, variable `x` can have different properties depending on which `if-else` clause executes.

```
function y = example2(use_fixpoint, data) %#codegen
    if use_fixpoint
        % x is fixed-point
        x = fi(data, 1, 12, 3);
    else
        % x is a matrix of doubles
        x = data;
    end
    % When x is reused here, it is not possible to determine its
    % class, size, and complexity
    t = sum(sum(x));
    y = t > 0;
end
```

Variable Reuse in an if Statement

To see how MATLAB renames a reused variable `t`:

- 1 Create a MATLAB file `example1.m` containing the following code.

```
function y = example1(u) %#codegen
    if all(all(u>0))
        % First, t is used to hold a scalar double value
        t = mean(mean(u)) / numel(u);
        u = u - t;
    end
    % t is reused to hold a vector of doubles
    t = find(u > 0);
```

```
y = sum(u(t(2:end-1)));
end
```

2 Compile example1.

For example, to generate a MEX function, enter:

```
codegen -o example1x -report example1.m -args {ones(5,5)}
```

Note `codegen` requires a MATLAB Coder license.

When the compilation is complete, `codegen` generates a MEX function, `example1x` in the current folder, and provides a link to the code generation report.

3 Open the code generation report.

4 In the MATLAB code pane of the code generation report, place your pointer over the variable `t` inside the `if` statement.

The code generation report highlights both instances of `t` in the `if` statement because they share the same class, size, and complexity. It displays the data type information for `t` at this point in the code. Here, `t` is a scalar double.

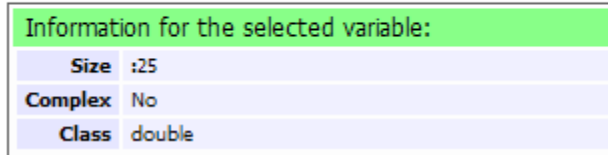
```
% First time t is used to hold a scalar double value.
t = mean(mean(u)) / numel(u);
u = u - t;
```

Information for the selected variable:	
Size	1 x 1
Complex	No
Class	double

5 In the MATLAB code pane of the report, place your pointer over the variable `t` outside the for-loop.

This time, the report highlights both instances of t outside the `if` statement. The report indicates that t might hold up to 25 doubles. The size of t is `:25`, that is, a column vector containing a maximum of 25 doubles.

```
t = find(u);  
y = sum(u(t(2:end-1)));
```



Information for the selected variable:	
Size	:25
Complex	No
Class	double

6 Click the **Variables** tab to view the list of variables used in `example1`.

The report displays a list of the variables in `example1`. There are two uniquely named local variables $t>1$ and $t>2$.

7 In the list of variables, place your pointer over $t>1$.

The code generation report highlights both instances of t in the `if` statement.

8 In the list of variables, place your pointer over $t>2$

The code generation report highlights both instances of t outside the `if` statement.

Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

- Persistent variables.
- Global variables.
- Variables passed to C code using `coder.ref`, `coder.rref`, `coder.wref`.
- Variables whose size is set using `coder.varsizes`.
- Variables whose names are controlled using `coder.cstructname`.
- The index variable of a `for`-loop when it is used inside the loop body.

- The block outputs of a MATLAB Function block in a Simulink model.
- Chart-owned variables of a MATLAB function in a Stateflow[®] chart.

Avoid Overflows in for-Loops

When memory integrity checks are enabled, if the code generation software detects that a loop variable might overflow on the last iteration of the for-loop, it reports an error.

To avoid this error, use the workarounds provided in the following table.

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"> • The loop counter increments by 1 • The end value equals the maximum value of the integer type • The loop is not covering the full range of the integer type 	<p>Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace:</p> <pre>N=intmax('int16') for k=N-10:N</pre> <p>with:</p> <pre>for k=1:10</pre>
<ul style="list-style-type: none"> • The loop counter decrements by 1 • The end value equals the minimum value of the integer type • The loop is not covering the full range of the integer type 	<p>Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace:</p> <pre>N=intmin('int32') for k=N+10:-1:N</pre> <p>with:</p> <pre>for k=10:-1:1</pre>

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"> • The loop counter increments or decrements by 1 • The start value equals the minimum or maximum value of the integer type • The end value equals the maximum or minimum value of the integer type <p>The loop covers the full range of the integer type.</p>	<p>Rewrite the loop casting the type of the loop counter start, step, and end values to a bigger integer or to double. For example, rewrite:</p> <pre>M= intmin('int16'); N= intmax('int16'); for k=M:N % Loop body end to M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N) % Loop body end</pre>
<ul style="list-style-type: none"> • The loop counter increments or decrements by a value not equal to 1 • On last loop iteration, the loop variable value is not equal to the end value <hr/> <p>Note The software error checking is conservative. It may incorrectly report a loop as being potentially infinite.</p> <hr/>	<p>Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.</p>

Supported Variable Types

You can use the following data types for C/C++ code generation from MATLAB:

Type	Description
char	Character array (string)
complex	Complex data. Cast function takes real and imaginary components
double	Double-precision floating point
int8, int16, int32	Signed integer
logical	Boolean true or false
single	Single-precision floating point
struct	Structure
uint8, uint16, uint32	Unsigned integer
Fixed-point	See “Fixed-Point Data Types”.

Defining Data for Code Generation

- “Data Definition for Code Generation” on page 6-2
- “Code Generation for Complex Data” on page 6-4
- “Code Generation for Characters” on page 6-6

Data Definition for Code Generation

To generate efficient standalone code, you must define the following types and classes of data differently than you normally would when running your code in the MATLAB environment:

Data	What's Different	More Information
Complex numbers	<ul style="list-style-type: none"> • Complexity of variables must be set at time of assignment and before first use • Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero <hr/> <p>Note Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic</p> <hr/>	“Code Generation for Complex Data” on page 6-4
Characters	Restricted to 8 bits of precision	“Code Generation for Characters” on page 6-6

Data	What's Different	More Information
Enumerated data	<ul style="list-style-type: none">• Supports integer-based enumerated types only• Restricted use in switch statements and for-loops	“Enumerated Data”
Function handles	<ul style="list-style-type: none">• Function handles must be scalar values• Same bound variable cannot reference different function handles• Cannot pass function handles to or from primary or extrinsic functions• Cannot view function handles from the debugger	“Function Handles”

Code Generation for Complex Data

In this section...

“Restrictions When Defining Complex Variables” on page 6-4

“Expressions Containing Complex Operands Yield Complex Results” on page 6-5

Restrictions When Defining Complex Variables

For code generation, you must set the complexity of variables at the time of assignment, either by assigning a complex constant or using the `complex` function, as in these examples:

```
x = 5 + 6i; % x is a complex number by assignment.  
y = 7 + 8j; % y is a complex number by assignment.  
x = complex(5,6); % x is the complex number 5 + 6i.
```

Once you set the type and size of a variable, you cannot cast it to another type or size. In the following example, the variable `x` is defined as complex and stays complex:

```
x = 1 + 2i; % Defines x as a complex variable.  
y = int16(x); % Real and imaginary parts of y are int16.  
x = 3; % x now has the value 3 + 0i.
```

Mismatches can also occur when you assign a real operand the complex result of an operation:

```
z = 3; % Sets type of z to double (real)  
z = 3 + 2i; % ERROR: cannot recast z to complex
```

As a workaround, set the complexity of the operand to match the result of the operation:

```
m = complex(3); % Sets m to complex variable of value 3 + 0i  
m = 5 + 6.7i; % Assigns a complex result to a complex number
```

Expressions Containing Complex Operands Yield Complex Results

In general, expressions that contain one or more complex operands produce a complex result in generated code, even if the value of the result is zero. Consider the following example:

```
x = 2 + 3i;  
y = 2 - 3i;  
z = x + y; % z is 4 + 0i.
```

In MATLAB, this code generates the real result $z = 4$. However, during code generation, the types for x and y are known, but their values are not. Because either or both operands in this expression are complex, z is defined as a complex variable requiring storage for both a real and an imaginary part. This means that z equals the complex result $4 + 0i$ in generated code, not 4 as in MATLAB code.

There are two exceptions to this behavior:

- Functions that take complex arguments, but produce real results

```
y = real(x); % y is the real part of the complex number x.  
y = imag(x); % y is the real-valued imaginary part of x.  
y = isreal(x); % y is false (0) for a complex number x.
```

- Functions that take real arguments, but produce complex results:

```
z = complex(x,y); % z is a complex number for a real x and y.
```

Code Generation for Characters

The complete set of Unicode® characters is not supported for code generation. Characters are restricted to 8 bits of precision in generated code. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with characters if you intend to generate code from your MATLAB algorithm.

Code Generation for Variable-Size Data

- “What Is Variable-Size Data?” on page 7-2
- “Variable-Size Data Definition for Code Generation” on page 7-3
- “Bounded Versus Unbounded Variable-Size Data” on page 7-4
- “Control Memory Allocation of Variable-Size Data” on page 7-5
- “Specify Variable-Size Data Without Dynamic Memory Allocation” on page 7-6
- “Variable-Size Data in Code Generation Reports” on page 7-10
- “Define Variable-Size Data for Code Generation” on page 7-12
- “C Code Interface for Arrays” on page 7-19
- “Diagnose and Fix Variable-Size Data Errors” on page 7-23
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 7-27
- “Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation” on page 7-34

What Is Variable-Size Data?

Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.

For example, in the following MATLAB function `nway`, `B` is a variable-size array; its length is not known at compile time.

```
function B = nway(A,n)
% Compute average of every N elements of A and put them in B.
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
        B(i) = mean(A(k + (0:n-1)));
        k = k + n;
    end
else
    error('n <= 0 or does not divide number of elements evenly');
end
```


Variable-Size Data Definition for Code Generation

In the MATLAB language, data can vary in size. By contrast, the semantics of generated code constrains the class, complexity, and shape of every expression, variable, and structure field. Therefore, for code generation, you must use each variable consistently. Each variable must:

- Be either complex or real (determined at first assignment)
- Have a consistent shape

For fixed-size data, the shape is the same as the size returned in MATLAB.

For example, if `size(A) == [4 5]`, the shape of variable A is 4 x 5.

For variable-size data, the shape can be abstract. That is, one or more dimensions can be unknown (such as `4x?` or `?x?`).

By default, the compiler detects code logic that attempts to change these fixed attributes after initial assignments, and flags these occurrences as errors during code generation. However, you can override this behavior by defining variables or structure fields as variable-size data. You can then generate standalone code for bounded and unbounded variable-size data.

For more information, see “Bounded Versus Unbounded Variable-Size Data” on page 7-4

Bounded Versus Unbounded Variable-Size Data

You can generate code for bounded and unbounded variable-size data. *Bounded variable-size data* has fixed upper bounds; this data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds; this data *must* be allocated on the heap. If you use unbounded data, you must use dynamic memory allocation so that the compiler:

- Does not check for upper bounds
- Allocates memory on the heap instead of the stack

You can control the memory allocation of variable-size data. For more information, see “Control Memory Allocation of Variable-Size Data” on page 7-5.

Control Memory Allocation of Variable-Size Data

Data whose size exceeds the dynamic memory allocation threshold is allocated on the heap. The default dynamic memory allocation threshold is 64 kilobytes. Data whose size is less than this threshold is allocated on the stack.

Dynamic memory allocation is an expensive operation; the performance cost might be too high for small data sets. If you use small variable-size data sets or data that does not change size at run time, disable dynamic memory allocation. See “Control Dynamic Memory Allocation” on page 19-99.

You can control memory allocation globally for your application by modifying the dynamic memory allocation threshold. See “Generate Code for a MATLAB Function That Expands a Vector in a Loop” on page 19-103. You can control memory allocation for individual variables by specifying upper bounds. See “Specifying Upper Bounds for Variable-Size Data” on page 7-6.

Specify Variable-Size Data Without Dynamic Memory Allocation

In this section...
“Fixing Upper Bounds Errors” on page 7-6
“Specifying Upper Bounds for Variable-Size Data” on page 7-6

Fixing Upper Bounds Errors

If MATLAB cannot determine or compute the upper bound, you must specify an upper bound. See “Specifying Upper Bounds for Variable-Size Data” on page 7-6 and “Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 7-25

Specifying Upper Bounds for Variable-Size Data

- “When to Specify Upper Bounds for Variable-Size Data” on page 7-6
- “Specifying Upper Bounds on the Command Line for Variable-Size Inputs” on page 7-6
- “Specifying Unknown Upper Bounds for Variable-Size Inputs” on page 7-7
- “Specifying Upper Bounds for Local Variable-Size Data” on page 7-7
- “Using a Matrix Constructor with Nonconstant Dimensions” on page 7-8

When to Specify Upper Bounds for Variable-Size Data

When using static allocation on the stack during code generation, MATLAB must be able to determine upper bounds for variable-size data. Specify the upper bounds explicitly for variable-size data from external sources, such as inputs and outputs.

Specifying Upper Bounds on the Command Line for Variable-Size Inputs

Use the `coder.typeof` construct with the `-args` option on the codegen command line (requires a MATLAB Coder license). For example:

```
codegen foo -args {coder.typeof(double(0),[3 100],1)}
```

This command specifies that the input to function `foo` is a matrix of real doubles with two variable dimensions. The upper bound for the first dimension is 3; the upper bound for the second dimension is 100. For a detailed explanation of this syntax, see `coder.typeof`.

Specifying Unknown Upper Bounds for Variable-Size Inputs

If you use dynamic memory allocation, you can specify that you don't know the upper bounds of inputs. To specify an unknown upper bound, use the infinity constant `Inf` in place of a numeric value. For example:

```
codegen foo -args {coder.typeof(double(0), [1 Inf])}
```

In this example, the input to function `foo` is a vector of real doubles without an upper bound.

Specifying Upper Bounds for Local Variable-Size Data

When using static allocation, MATLAB uses a sophisticated analysis to calculate the upper bounds of local data at compile time. However, when the analysis fails to detect an upper bound or calculates an upper bound that is not precise enough for your application, you need to specify upper bounds explicitly for local variables.

You do not need to specify upper bounds when using dynamic allocation on the heap. In this case, MATLAB assumes variable-size data is unbounded and does not attempt to determine upper bounds.

Constraining the Value of a Variable That Specifies Dimensions of Variable-Size Data. Use the `assert` function with relational operators to constrain the value of variables that specify the dimensions of variable-size data. For example:

```
function y = dim_need_bound(n) %#codegen
assert (n <= 5);
L = ones(n,n);
M = zeros(n,n);
M = [L; M];
y = M;
```

This `assert` statement constrains input `n` to a maximum size of 5, defining `L` and `M` as variable-sized matrices with upper bounds of 5 for each dimension.

Specifying the Upper Bounds for All Instances of a Local Variable.

Use the `coder.varsize` function to specify the upper bounds for all instances of a local variable in a function. For example:

```
function Y = example_bounds1(u) %#codegen
Y = [1 2 3 4 5];
coder.varsize('Y', [1 10]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

The second argument of `coder.varsize` specifies the upper bound for each instance of the variable specified in the first argument. In this example, the argument `[1 10]` indicates that for every instance of `Y`:

- First dimension is fixed at size 1
- Second dimension can grow to an upper bound of 10

By default, `coder.varsize` assumes dimensions of 1 are fixed size. For more information, see the `coder.varsize` reference page.

Using a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For example:

```
function y = var_by_assign(u) %#codegen
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

If you are not using dynamic memory allocation, you must also add an `assert` statement to provide upper bounds for the dimensions. For example:

```
function y = var_by_assign(u) %#codegen
assert (u < 20);
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

Variable-Size Data in Code Generation Reports

In this section...
“What Reports Tell You About Size” on page 7-10
“How Size Appears in Code Generation Reports” on page 7-11
“How to Generate a Code Generation Report” on page 7-11

What Reports Tell You About Size

Code generation reports:

- Differentiate fixed-size from variable-size data
- Identify variable-size data with unknown upper bounds
- Identify variable-size data with fixed dimensions

If you define a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends * to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions does not change during execution.

- Provide guidance on how to fix size mismatch and upper bounds errors.

How Size Appears in Code Generation Reports

:? means variable size, unknown upper bound

Variable	Type	Size
B	Output	1 x :?
A	Input	1 x :100
n	Input	1 x 1

No colon prefix (:) means fixed size

:100 means variable size, upper bound = 100

Variable	Type	Size
y	Output	1 x 10*

* means that you declared y as variable size, but subsequently fixed its dimensions

How to Generate a Code Generation Report

Add the -report option to your codegen command.

Define Variable-Size Data for Code Generation

In this section...

“When to Define Variable-Size Data Explicitly” on page 7-12

“Using a Matrix Constructor with Nonconstant Dimensions” on page 7-13

“Inferring Variable Size from Multiple Assignments” on page 7-13

“Defining Variable-Size Data Explicitly Using `coder.varsize`” on page 7-14

When to Define Variable-Size Data Explicitly

For code generation, you must assign variables to have a specific class, size, and complexity before using them in operations or returning them as outputs. Generally, you cannot reassign variable properties after the initial assignment. Therefore, attempts to grow a variable or structure field after assigning it a fixed size might cause a compilation error. In these cases, you must explicitly define the data as variable sized using one of these methods:

Method	See
Assign the data from a variable-size matrix constructor such as <ul style="list-style-type: none"> • <code>ones</code> • <code>zeros</code> • <code>repmat</code> 	“Using a Matrix Constructor with Nonconstant Dimensions” on page 7-13
Assign multiple, constant sizes to the same variable before using (reading) the variable.	“Inferring Variable Size from Multiple Assignments” on page 7-13
Define all instances of a variable to be variable sized	“Defining Variable-Size Data Explicitly Using <code>coder.varsize</code> ” on page 7-14

Using a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For example:

```
function y = var_by_assign(u) %#codegen
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

Inferring Variable Size from Multiple Assignments

You can define variable-size data by assigning multiple, constant sizes to the same variable before you use (read) the variable in your code. When MATLAB uses static allocation on the stack for code generation, it infers the upper bounds from the largest size specified for each dimension. When you assign the same size to a given dimension across all assignments, MATLAB assumes that the dimension is fixed at that size. The assignments can specify different shapes as well as sizes.

When dynamic memory allocation is used, MATLAB does not check for upper bounds; it assumes variable-size data is unbounded.

Inferring Upper Bounds from Multiple Definitions with Different Shapes

```
function y = var_by_multiassign(u) %#codegen
if (u > 0)
    y = ones(3,4,5);
else
    y = zeros(3,1);
end
```

When static allocation is used, this function infers that `y` is a matrix with three dimensions, where:

- First dimension is fixed at size 3

- Second dimension is variable with an upper bound of 4
- Third dimension is variable with an upper bound of 5

The code generation report represents the size of matrix `y` like this:

Variable	Type	Size
<code>y</code>	Output	<code>3 x :4 x :5</code>

When dynamic allocation is used, the function analyzes the dimensions of `y` differently:

- First dimension is fixed at size 3
- Second and third dimensions are unbounded

In this case, the code generation report represents the size of matrix `y` like this:

Variable	Type	Size
<code>y</code>	Output	<code>3 x :? x :?</code>

Defining Variable-Size Data Explicitly Using `coder.varsize`

Use the function `coder. varsize` to define one or more variables or structure fields as variable-size data. Optionally, you can also specify which dimensions vary along with their upper bounds (see “Specifying Which Dimensions Vary” on page 7-15). For example:

- Define `B` as a variable-size 2-by-2 matrix, where each dimension has an upper bound of 64:

```
coder. varsize('B', [64 64]);
```

- Define `B` as a variable-size matrix:

```
coder.varsize('B');
```

When you supply only the first argument, `coder.varsize` assumes all dimensions of `B` can vary and that the upper bound is `size(B)`.

For more information, see the `coder.varsize` reference page.

Specifying Which Dimensions Vary

You can use the function `coder.varsize` to specify which dimensions vary. For example, the following statement defines `B` as a row vector whose first dimension is fixed at 2, but whose second dimension can grow to an upper bound of 16:

```
coder.varsize('B', [2, 16], [0 1])
```

The third argument specifies which dimensions vary. This argument must be a logical vector or a double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` have fixed size; dimensions that correspond to ones or `true` vary in size. `coder.varsize` usually treats dimensions of size 1 as fixed (see “Defining Variable-Size Matrices with Singleton Dimensions” on page 7-16).

For more information about the syntax, see the `coder.varsize` reference page.

Allowing a Variable to Grow After Defining Fixed Dimensions

Function `var_by_if` defines matrix `Y` with fixed 2-by-2 dimensions before first use (where the statement `Y = Y + u` reads from `Y`). However, `coder.varsize` defines `Y` as a variable-size matrix, allowing it to change size based on decision logic in the `else` clause:

```
function Y = var_by_if(u) %#codegen
if (u > 0)
    Y = zeros(2,2);
    coder.varsize('Y');
    if (u < 10)
        Y = Y + u;
    end
else
```

```
    Y = zeros(5,5);  
end
```

Without `coder. varsize`, MATLAB infers `Y` to be a fixed-size, 2-by-2 matrix and generates a size mismatch error during code generation.

Defining Variable-Size Matrices with Singleton Dimensions

A singleton dimension is a dimension for which `size(A,dim) = 1`. Singleton dimensions are fixed in size when:

- You specify a dimension with an upper bound of 1 in `coder. varsize` expressions.

For example, in this function, `Y` behaves like a vector with one variable-size dimension:

```
function Y = dim_singleton(u) %#codegen  
Y = [1 2];  
coder. varsize('Y', [1 10]);  
if (u > 0)  
    Y = [Y 3];  
else  
    Y = [Y u];  
end
```

- You initialize variable-size data with singleton dimensions using matrix constructor expressions or matrix functions.

For example, in this function, both `X` and `Y` behave like vectors where only their second dimensions are variable sized:

```
function [X,Y] = dim_singleton_vects(u) %#codegen  
Y = ones(1,3);  
X = [1 4];  
coder. varsize('Y','X');  
if (u > 0)  
    Y = [Y u];  
else  
    X = [X u];  
end
```

You can override this behavior by using `coder.versize` to specify explicitly that singleton dimensions vary. For example:

```
function Y = dim_singleton_vary(u) %#codegen
Y = [1 2];
coder.versize('Y', [1 10], [1 1]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

In this example, the third argument of `coder.versize` is a vector of ones, indicating that each dimension of `Y` varies in size. For more information, see the `coder.versize` reference page.

Defining Variable-Size Structure Fields

To define structure fields as variable-size arrays, use colon (`:`) as the index expression. The colon (`:`) indicates that all elements of the array are variable sized. For example:

```
function y=struct_example() %#codegen

d = struct('values', zeros(1,0), 'color', 0);
data = repmat(d, [3 3]);
coder.versize('data(:).values');

for i = 1:numel(data)
    data(i).color = rand-0.5;
    data(i).values = 1:i;
end

y = 0;
for i = 1:numel(data)
    if data(i).color > 0
        y = y + sum(data(i).values);
    end;
end
```

The expression `coder.ysize('data(:).values')` defines the field values inside each element of matrix `data` to be variable sized.

Here are other examples:

- `coder.ysize('data.A(:).B')`

In this example, `data` is a scalar variable that contains matrix `A`. Each element of matrix `A` contains a variable-size field `B`.

- `coder.ysize('data(:).A(:).B')`

This expression defines field `B` inside each element of matrix `A` inside each element of matrix `data` to be variable sized.

C Code Interface for Arrays

In this section...

“C Code Interface for Statically Allocated Arrays” on page 7-19

“C Code Interface for Dynamically Allocated Arrays” on page 7-20

“Utility Functions for Creating emxArray Data Structures” on page 7-21

C Code Interface for Statically Allocated Arrays

In generated code, MATLAB contains two pieces of information about statically allocated arrays: the maximum size of the array and its actual size.

For example, consider the MATLAB function `uniquetol`:

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B');
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

Generate code for `uniquetol` specifying that input `A` is a variable-size real double vector whose first dimension is fixed at 1 and second dimension can vary up to 100 elements.

```
codegen -config:lib -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

In the generated code, the function declaration is:

```
extern void uniquetol(const real_T A_data[100], const int32_T A_size[2],...
    real_T tol, emxArray_real_T *B);
```

There are two pieces of information about A:

- `real_T A_data[100]`: the maximum size of input A (where 100 is the maximum size specified using `coder.typeof`).
- `int32_T A_sizes[2]`: the actual size of the input.

C Code Interface for Dynamically Allocated Arrays

In generated code, MATLAB represents dynamically allocated data as a structure type called `emxArray`. An embeddable version of the MATLAB `mxArray`, the `emxArray` is a family of data types, specialized for all base types.

emxArray Structure Definition

```
typedef struct emxArray_<baseTypeName>
{
    <baseTypeName> *data;
    int32_T *size;
    int32_T allocated;
    int32_T numDimensions;
    boolean_T canFreeData;
} emxArray_<baseTypeName>;
```

For example, here's the definition for an `emxArray` of base type `real_T` with unknown upper bounds:

```
typedef struct emxArray_real_T
{
    real_T *data;
    int32_T *size;
    int32_T allocated;
    int32_T numDimensions;
    boolean_T canFreeData;
} emxArray_real_T;
```

To define two variables, `in1` and `in2`, of this type, use this statement:

```
emxArray_real_T *in1, *in2;
```

C Code Interface for Structure Fields

Field	Description
*data	Pointer to data of type <i><baseTypeName></i>
*size	Pointer to first element of size vector. Length of the vector equals the number of dimensions.
allocatedSize	Number of elements currently allocated for the array. If the size changes, MATLAB reallocates memory based on the new size.
numDimensions	Number of dimensions of the size vector, that is, the number of dimensions you can access without crossing into unallocated or unused memory
canFreeData	Boolean flag indicating how to deallocate memory: <ul style="list-style-type: none"> • <code>true</code> – MATLAB deallocates memory automatically • <code>false</code> – Calling program determines when to deallocate memory

Utility Functions for Creating emxArray Data Structures

When you generate code that uses variable-size data, the code generation software exports a set of utility functions that you can use to create and interact with `emxArrays` in your generated code. To call these functions in your main C function, include the generated header file. For example, when you generate code for function `foo`, include `foo_emxAPI.h` in your main C function. For more information, see the “Write a C Main Function” section in “Using Dynamic Memory Allocation for an "Atoms" Simulation” on page 19-110.

Function	Arguments	Description
<code>emxArray_<baseTypeName></code> <code>*emxCreateWrapper_<baseTypeName></code> <code>(...)</code>	<code>*data</code> <code>num_rows</code> <code>num_cols</code>	Creates a new 2-dimensional <code>emxArray</code> , but does not allocate it on the heap. Instead uses memory provided by the user and sets <code>canFreeData</code> to <code>false</code> so it does not inadvertently free user memory, such as the stack.
<code>emxArray_<baseTypeName></code> <code>*emxCreateWrapperND_<baseTypeName></code> <code>(...)</code>	<code>*data</code> <code>numDimensions</code> <code>*size</code>	Same as <code>emxCreateWrapper</code> , except it creates a new N-dimensional <code>emxArray</code> .
<code>emxArray_<baseTypeName></code> <code>*emxCreate_<baseTypeName> (...)</code>	<code>num_rows</code> <code>num_cols</code>	Creates a new two-dimensional <code>emxArray</code> on the heap, initialized to zero. All data elements have the data type specified by <i>baseTypeName</i> .
<code>emxArray_<baseTypeName></code> <code>*emxCreateND_<baseTypeName> (...)</code>	<code>numDimensions</code> <code>*size</code>	Same as <code>emxCreate</code> , except it creates a new N-dimensional <code>emxArray</code> on the heap.
<code>emxArray_<baseTypeName></code> <code>*emxDestroyArray_<baseTypeName></code> <code>(...)</code>	<code>*emxArray</code>	Frees dynamic memory allocated by <code>*emxCreate</code> and <code>*emxCreateND</code> functions.

Diagnose and Fix Variable-Size Data Errors

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 7-23

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 7-25

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

You cannot assign variable-size matrices to fixed-size matrices in generated code. Consider this example:

```
function Y = example_mismatch1(n) %#codegen
assert(n<10);
B = ones(n,n);
A = magic(3);
A(1) = mean(A(:));
if (n == 3)
    A = B;
end
Y = A;
```

Compiling this function produces this error:

```
??? Dimension 1 is fixed on the left-hand side
but varies on the right ...
```

There are several ways to fix this error:

- Allow matrix A to grow by adding the `coder. varsize` construct:

```
function Y = example_mismatch1_fix1(n) %#codegen
coder. varsize('A');
assert(n<10);
B = ones(n,n);
A = magic(3);
```

```
A(1) = mean(A(:));  
if (n == 3)  
    A = B;  
end  
Y = A;
```

- Explicitly restrict the size of matrix B to 3-by-3 by modifying the assert statement:

```
function Y = example_mismatch1_fix2(n) %#codegen  
coder.varsize('A');  
assert(n==3)  
B = ones(n,n);  
A = magic(3);  
A(1) = mean(A(:));  
if (n == 3)  
    A = B;  
end  
Y = A;
```

- Use explicit indexing to make B the same size as A:

```
function Y = example_mismatch1_fix3(n) %#codegen  
assert(n<10);  
B = ones(n,n);  
A = magic(3);  
A(1) = mean(A(:));  
if (n == 3)  
    A = B(1:3, 1:3);  
end  
Y = A;
```

Empty Matrix Reshaped to Match Variable-Size Specification

If you assign an empty matrix [] to variable-size data, MATLAB might silently reshape the data in generated code to match a `coder.varsize` specification. For example:

```
function Y = test(u) %#codegen  
Y = [];  
coder.varsize('Y', [1 10]);
```

```

If u < 0
    Y = [Y u];
end

```

In this example, `coder. varsize` defines `Y` as a column vector of up to 10 elements, so its first dimension is fixed at size 1. The statement `Y = []` designates the first dimension of `Y` as 0, creating a mismatch. The right hand side of the assignment is an empty matrix and the left hand side is a variable-size vector. In this case, MATLAB reshapes the empty matrix `Y = []` in generated code to `Y = zeros(1,0)` so it matches the `coder. varsize` specification.

Performing Binary Operations on Fixed and Variable-Size Operands

You cannot perform binary operations on operands of different sizes. Operands have different sizes if one has fixed dimensions and the other has variable dimensions. For example:

```

function z = mismatch_operands(n) %#codegen
    assert(n>=3 && n<10);
    x = ones(n,n);
    y = magic(3);
    z = x + y;

```

When you compile this function, you get an error because `y` has fixed dimensions (3 x 3), but `x` has variable dimensions. Fix this problem by using explicit indexing to make `x` the same size as `y`:

```

function z = mismatch_operands_fix(n) %#codegen
    assert(n>=3 && n<10);
    x = ones(n,n);
    y = magic(3);
    z = x(1:3,1:3) + y;

```

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

You can define variable-size data by assigning a variable to a matrix with nonconstant dimensions. For example:

```
function y = dims_vary(u) %#codegen
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

However, compiling this function generates an error because you did not specify an upper bound for `u`.

There are several ways to fix the problem:

- Enable dynamic memory allocation and recompile. During code generation, MATLAB does not check for upper bounds when it uses dynamic memory allocation for variable-size data.
- If you do not want to use dynamic memory allocation, add an `assert` statement before the first use of `u`:

```
function y = dims_vary_fix(u) %#codegen
assert (u < 20);
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```


Incompatibilities with MATLAB in Variable-Size Support for Code Generation

In this section...

“Incompatibility with MATLAB for Scalar Expansion” on page 7-27

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-29

“Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 7-30

“Incompatibility with MATLAB in Vector-Vector Indexing” on page 7-31

“Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-32

“Incompatibility with MATLAB in Concatenating Variable-Size Matrices” on page 7-33

“Dynamic Memory Allocation Not Supported for MATLAB Function Blocks” on page 7-33

Incompatibility with MATLAB for Scalar Expansion

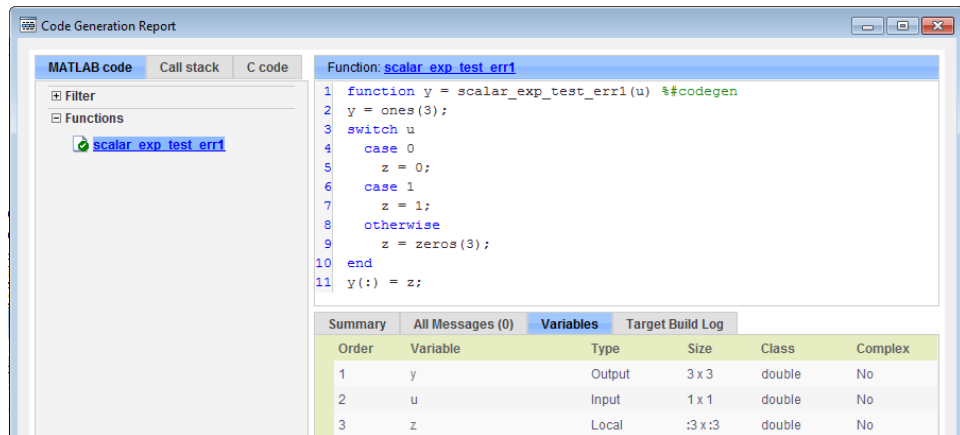
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

During code generation, the standard MATLAB scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Instead, it generates a size mismatch error at run time for MEX functions. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior.

For example, in the following function, `z` is scalar for the `switch` statement case 0 and case 1. MATLAB applies scalar expansion when evaluating `y(:) = z;` for these two cases.

```
function y = scalar_exp_test_err1(u) %#codegen
for the otherwise case of the switch function.y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z;
```

When you generate code for this function, the code generation software determines that `z` is variable size with an upper bound of 3.



The screenshot shows the 'Code Generation Report' window. The 'MATLAB code' tab is active, displaying the source code for the function `scalar_exp_test_err1`. Below the code, the 'Variables' tab is selected, showing a summary table of variables.

Order	Variable	Type	Size	Class	Complex
1	y	Output	3 x 3	double	No
2	u	Input	1 x 1	double	No
3	z	Local	:3 x:3	double	No

If you run the MEX function with `u` equal to zero or one, even though `z` is scalar at run time, the generated code does not perform scalar expansion and a run-time error occurs.

```
scalar_exp_test_err1_mex(0)
Sizes mismatch: 9 ~= 1.
```

```
Error in scalar_exp_test_err1 (line 11)
y(:) = z;
```

Workaround

Use indexing to force z to be a scalar value:

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z(1);
```

Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function can return a different result in generated code than in MATLAB. In generated code, `size(A)` returns a fixed-length output because it does not drop trailing singleton dimensions of variable-size N-D arrays. By contrast, `size(A)` in MATLAB returns a variable-length output because it drops trailing singleton dimensions.

For example, if the shape of array A is `?x:?x:?` and `size(A,3)==1`, `size(A)` returns:

- Three-element vector in generated code
- Two-element vector in MATLAB code

Workarounds

If your application requires generated code to return the same size of variable-size N-D arrays as MATLAB code, consider one of these workarounds:

- Use the two-argument form of `size`.

For example, `size(A,n)` returns the same answer in generated code and MATLAB code.

- Rewrite `size(A)`:

```
B = size(A);  
X = B(1:ndims(A));
```

This version returns `X` with a variable-length output. However, you cannot pass a variable-size `X` to matrix constructors such as `zeros` that require a fixed-size argument.

Incompatibility with MATLAB in Determining Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. The size might be `1x0` or `0x1` in generated code, but `0x0` in MATLAB. Therefore, you should not write code that relies on the specific size of empty matrices.

For example, consider the following code:

```
function y = foo(n) %#codegen  
x = [];  
i=0;  
    while (i<10)  
        x = [5, x];  
        i=i+1;  
    end  
if n > 0  
    x = [];  
end  
y=size(x);  
end
```

Concatenation requires its operands to match on the size of the dimension that is not being concatenated. In the preceding concatenation the scalar value has size `1x1` and `x` has size `0x0`. To support this use case, the code generation software determines the size for `x` as `[1 x :?]`. Because there

is another assignment `x = []` after the concatenation, the size of `x` in the generated code is `1x0` instead of `0x0`.

Workaround

If your application checks whether a matrix is empty, use one of these workarounds:

- Rewrite your code to use the `isempty` function instead of the `size` function.
- Instead of using `x=[]` to create empty arrays, create empty arrays of a specific size using `zeros`. For example:

```
function y = test_empty(n) %#codegen
x = zeros(1,0);
i=0;
    while (i<10)
        x = [5, x];
        i=i+1;
    end
if n > 0
    x = zeros(1,0);
end
y=size(x);
end
```

Incompatibility with MATLAB in Vector-Vector Indexing

In vector-vector indexing, you use one vector as an index into another vector. When either vector is variable sized, you might get a run-time error during code generation. Consider the index expression `A(B)`. The general rule for indexing is that `size(A(B)) == size(B)`. However, when both `A` and `B` are vectors, MATLAB applies a special rule: use the orientation of `A` as the orientation of the output. For example, if `size(A) == [1 5]` and `size(B) == [3 1]`, then `size(A(B)) == [1 3]`.

In this situation, if the code generation software detects that both `A` and `B` are vectors at compile time, it applies the special rule and gives the same result as MATLAB. However, if either `A` or `B` is a variable-size matrix (has shape `?x?`) at compile time, the code generation software applies only the general

indexing rule. Then, if both A and B become vectors at run time, the code generation software reports a run-time error when you run the MEX function. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior. It is best practice to generate and test a MEX function before generating C code.

Workaround

Force your data to be a vector by using the colon operator for indexing: `A(B(:))`. For example, suppose your code intentionally toggles between vectors and regular matrices at run time. You can do an explicit check for vector-vector indexing:

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
    D = A(B);
end
...
```

The indexing in the first branch specifies that `C` and `B(:)` are compile-time vectors. As a result, the code generation software applies the standard vector-vector indexing rule.

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

The following limitation applies to matrix indexing operations for code generation:

- Initialization of the following style:

```
for i = 1:10
    M(i) = 5;
end
```

In this case, the size of `M` changes as the loop is executed. Code generation does not support increasing the size of an array over time.

For code generation, preallocate M as highlighted in the following code.

```
M=zeros(1,10);  
for i = 1:10  
    M(i) = 5;  
end
```

The following limitation applies to matrix indexing operations for code generation when dynamic memory allocation is disabled:

- $M(i:j)$ where i and j change in a loop

During code generation, memory is not dynamically allocated for the size of the expressions that change as the program executes. To implement this behavior, use for-loops as shown in the following example:

```
...  
M = ones(10,10);  
for i=1:10  
    for j = i:10  
        M(i,j) = 2 * M(i,j);  
    end  
end  
...
```

Note The matrix M must be defined before entering the loop, as shown in the highlighted code.

Incompatibility with MATLAB in Concatenating Variable-Size Matrices

For code generation, when you concatenate variable-sized arrays, the dimensions that are not being concatenated must match exactly.

Dynamic Memory Allocation Not Supported for MATLAB Function Blocks

You cannot use dynamic memory allocation for variable-size data in MATLAB Function blocks. Use bounded instead of unbounded variable-size data.

Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation

In this section...

“Common Restrictions” on page 7-34

“Toolbox Functions with Variable Sizing Restrictions” on page 7-35

Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in “Toolbox Functions with Variable Sizing Restrictions” on page 7-35.

Variable-length vector restriction

Inputs to the library function must be variable-length vectors or fixed-size vectors. A variable-length vector is a variable-size array that has the shape $1 \times n$ or $n \times 1$ (one dimension is variable sized and the other is fixed at size 1). Other shapes are not permitted, even if they are vectors at run time.

Automatic dimension restriction

When the function selects the working dimension automatically, it bases the selection on the upper bounds for the dimension sizes. In the case of the `sum` function, `sum(X)` selects its working dimension automatically, while `sum(X, dim)` uses `dim` as the explicit working dimension.

For example, if `X` is a variable-size matrix with dimensions $1 \times 3 \times 5$, `sum(x)` behaves like `sum(X,2)` in generated code. In MATLAB, it behaves like `sum(X,2)` provided `size(X,2)` is not 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`. Consequently, you get a run-time error if an automatically selected working dimension assumes a length of 1 at run time.

To avoid the issue, specify the intended working dimension explicitly as a constant value.

Array-to-vector restriction

The function issues an error when a variable-size array that is not a variable-length vector assumes the shape of a vector at run time. To avoid the issue, specify the input explicitly as a variable-length vector instead of a variable-size array.

Array-to-scalar restriction

The function issues an error if a variable-size array assumes a scalar value at run time. To avoid this issue, specify scalars as fixed size.

Toolbox Functions with Variable Sizing Restrictions

The following restrictions apply to specific toolbox functions, but only for code generation.

Function	Restrictions with Variable-Size Data
all	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
any	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
bsxfun	<ul style="list-style-type: none"> • Dimensions expand only where one input array or the other has a fixed length of 1.
cat	<ul style="list-style-type: none"> • Dimension argument must be a constant. • An error occurs if variable-size inputs are empty at run time.

Function	Restrictions with Variable-Size Data
conv	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-34. • Input vectors must have the same orientation, either both row vectors or both column vectors.
cov	<ul style="list-style-type: none"> • For cov(X), see “Array-to-vector restriction” on page 7-35.
cross	<ul style="list-style-type: none"> • Variable-size array inputs that become vectors at run time must have the same orientation.
deconv	<ul style="list-style-type: none"> • For both arguments, see “Variable-length vector restriction” on page 7-34.
detrend	<ul style="list-style-type: none"> • For first argument for row vectors only, see “Array-to-vector restriction” on page 7-35 .
diag	<ul style="list-style-type: none"> • See “Array-to-vector restriction” on page 7-35 .
diff	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • Length of the working dimension must be greater than the difference order input when the input is variable sized. For example, if the input is a variable-size matrix that is 3-by-5 at run time, diff(x,2,1) works but diff(x,5,1) generates a run-time error.
fft	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34.

Function	Restrictions with Variable-Size Data
filter	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 7-34. • See “Automatic dimension restriction” on page 7-34.
hist	<ul style="list-style-type: none"> • For second argument, see “Variable-length vector restriction” on page 7-34. • For second input argument, see “Array-to-scalar restriction” on page 7-35.
histc	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34.
ifft	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34.
ind2sub	<ul style="list-style-type: none"> • First input (the size vector input) must be fixed size.
interp1	<ul style="list-style-type: none"> • For the Y input and xi input, see “Array-to-vector restriction” on page 7-35. • Y input can become a column vector dynamically. • A run-time error occurs if Y input is not a variable-length vector and becomes a row vector at run time.
ipermute	<ul style="list-style-type: none"> • Order input must be fixed size.
issorted	<ul style="list-style-type: none"> • For optional rows input, see “Variable-length vector restriction” on page 7-34.

Function	Restrictions with Variable-Size Data
magic	<ul style="list-style-type: none">• Argument must be a constant.• Output can be fixed-size matrices only.
max	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 7-34.
mean	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 7-34.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
median	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 7-34.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
min	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 7-34.
mode	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 7-34.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
mtimes	<ul style="list-style-type: none"> • When an input is variable sized, MATLAB determines whether to generate code for a general matrix*matrix multiplication or a scalar*matrix multiplication, based on whether one of the arguments is a fixed-size scalar. If neither argument is a fixed-size scalar, the inner dimensions of the two arguments must agree even if a variable-size matrix input happens to be a scalar at run time.
nchoosek	<ul style="list-style-type: none"> • Inputs must be fixed sized. • Second input must be a constant for static allocation. If you enable dynamic allocation, second input can be a variable. • You cannot create a variable-size array by passing in a variable k unless you enable dynamic allocation.
permute	<ul style="list-style-type: none"> • Order input must be fixed size.
planerot	<ul style="list-style-type: none"> • Input must be a fixed-size, two-element column vector. It cannot be a variable-size array that takes on the size 2-by-1 at run time.
poly	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-34.
polyfit	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 7-34.

Function	Restrictions with Variable-Size Data
prod	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
rand	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>rand(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>rand([1,N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
randn	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>randn(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>randn([1,N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
reshape	<ul style="list-style-type: none"> • When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.
roots	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-34.

Function	Restrictions with Variable-Size Data
shiftdim	<ul style="list-style-type: none"> • If you do not supply the second argument, the number of shifts is determined at compilation time by the upper bounds of the dimension sizes. Consequently, at run time the number of shifts is constant. • An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant). • First input argument must have the same number of dimensions when you supply a positive number of shifts.
std	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
sub2ind	<ul style="list-style-type: none"> • First input (the size vector input) must be fixed size.
sum	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
trapz	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-34. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
typecast	<ul style="list-style-type: none">• See “Variable-length vector restriction” on page 7-34 on first argument.
var	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 7-34.• An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

Code Generation for MATLAB Structures

- “Structure Definition for Code Generation” on page 8-2
- “Structure Operations Allowed for Code Generation” on page 8-3
- “Define Scalar Structures for Code Generation” on page 8-4
- “Define Arrays of Structures for Code Generation” on page 8-7
- “Make Structures Persistent” on page 8-9
- “Index Substructures and Fields” on page 8-10
- “Assign Values to Structures and Fields” on page 8-12
- “Pass Large Structures as Input Parameters” on page 8-13

Structure Definition for Code Generation

To generate efficient standalone code for structures, you must define and use structures differently than you normally would when running your code in the MATLAB environment:

What's Different	More Information
Use a restricted set of operations.	“Structure Operations Allowed for Code Generation” on page 8-3
Observe restrictions on properties and values of scalar structures.	“Define Scalar Structures for Code Generation” on page 8-4
Make structures uniform in arrays.	“Define Arrays of Structures for Code Generation” on page 8-7
Reference structure fields individually during indexing.	“Index Substructures and Fields” on page 8-10
Avoid type mismatch when assigning values to structures and fields.	“Assign Values to Structures and Fields” on page 8-12

Structure Operations Allowed for Code Generation

To generate efficient standalone code for MATLAB structures, you are restricted to the following operations:

- Define structures as local and persistent variables by assignment and using the `struct` function
- Index structure fields using dot notation
- Define primary function inputs as structures
- Pass structures to local functions

Define Scalar Structures for Code Generation

In this section...

“Restriction When Using struct” on page 8-4

“Restrictions When Defining Scalar Structures by Assignment” on page 8-4

“Adding Fields in Consistent Order on Each Control Flow Path” on page 8-4

“Restriction on Adding New Fields After First Use” on page 8-5

Restriction When Using struct

When you use the `struct` function to create scalar structures for code generation, you cannot create structures of cell arrays.

Restrictions When Defining Scalar Structures by Assignment

When you define a scalar structure by assigning a variable to a preexisting structure, you do not need to define the variable before the assignment. However, if you already defined that variable, it must have the same class, size, and complexity as the structure you assign to it. In the following example, `p` is defined as a structure that has the same properties as the predefined structure `S`:

```
...  
S = struct('a', 0, 'b', 1, 'c', 2);  
p = S;  
...
```

Adding Fields in Consistent Order on Each Control Flow Path

When you create a structure, you must add fields in the same order on each control flow path. For example, the following code generates a compiler error because it adds the fields of structure `x` in a different order in each `if` statement clause:

```
function y = fcn(u) %#codegen  
if u > 0
```

```

    x.a = 10;
    x.b = 20;
else
    x.b = 30; % Generates an error (on variable x)
    x.a = 40;
end
y = x.a + x.b;

```

In this example, the assignment to `x.a` comes before `x.b` in the first `if` statement clause, but the assignments appear in reverse order in the `else` clause. Here is the corrected code:

```

function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.a = 40;
    x.b = 30;
end
y = x.a + x.b;

```

Restriction on Adding New Fields After First Use

You cannot add fields to a structure after you perform the following operations on the structure:

- Reading from the structure
- Indexing into the structure array
- Passing the structure to a function

For example, consider this code:

```

...
x.c = 10; % Defines structure and creates field c
y = x; % Reads from structure
x.d = 20; % Generates an error
...

```

In this example, the attempt to add a new field `d` after reading from structure `x` generates an error.

This restriction extends across the structure hierarchy. For example, you cannot add a field to a structure after operating on one of its fields or nested structures, as in this example:

```
function y = fcn(u) %#codegen

x.c = 10;
y = x.c;
x.d = 20; % Generates an error
```

In this example, the attempt to add a new field `d` to structure `x` after reading from the structure's field `c` generates an error.

Define Arrays of Structures for Code Generation

In this section...

“Ensuring Consistency of Fields” on page 8-7

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 8-7

“Defining an Array of Structures Using Concatenation” on page 8-8

Ensuring Consistency of Fields

When you create an array of MATLAB structures with the intent of generating code, you must be sure that each structure field in the array has the same size, type, and complexity.

Using repmat to Define an Array of Structures with Consistent Field Properties

You can create an array of structures from a scalar structure by using the MATLAB repmat function, which replicates and tiles an existing scalar structure:

- 1 Create a scalar structure, as described in “Define Scalar Structures for Code Generation” on page 8-4.
- 2 Call repmat, passing the scalar structure and the dimensions of the array.
- 3 Assign values to each structure using standard array indexing and structure dot notation.

For example, the following code creates X, a 1-by-3 array of scalar structures. Each element of the array is defined by the structure s, which has two fields, a and b:

```
...  
s.a = 0;  
s.b = 0;  
X = repmat(s,1,3);  
X(1).a = 1;
```

```
X(2).a = 2;  
X(3).a = 3;  
X(1).b = 4;  
X(2).b = 5;  
X(3).b = 6;  
...
```

Defining an Array of Structures Using Concatenation

To create a small array of structures, you can use the concatenation operator, square brackets ([]), to join one or more structures into an array (see “Concatenating Matrices”). For code generation, the structures that you concatenate must have the same size, class, and complexity.

For example, the following code uses concatenation and a local function to create the elements of a 1-by-3 structure array:

```
...  
W = [ sab(1,2) sab(2,3) sab(4,5) ];  
  
function s = sab(a,b)  
    s.a = a;  
    s.b = b;  
...
```


Make Structures Persistent

To make structures persist, you define them to be persistent variables and initialize them with the `isempty` statement, as described in “Define and Initialize Persistent Variables” on page 5-10.

For example, the following function defines structure `X` to be persistent and initializes its fields `a` and `b`:

```
function f(u) %#codegen
persistent X

if isempty(X)
    X.a = 1;
    X.b = 2;
end
```

Index Substructures and Fields

Use these guidelines when indexing substructures and fields for code generation:

Reference substructure field values individually using dot notation

For example, the following MATLAB code uses dot notation to index fields and substructures:

```
...
substruct1.a1 = 15.2;
substruct1.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                 'ele3',substruct1);

substruct2 = mystruct;
substruct2.ele3.a2 = 2*(substruct1.a2);
...
```

The generated code indexes elements of the structures in this example by resolving symbols as follows:

Dot Notation	Symbol Resolution
substruct1.a1	Field a1 of local structure substruct1
substruct2.ele3.a1	Value of field a1 of field ele3, a substructure of local structure substruct2
substruct2.ele3.a2(1,1)	Value in row 1, column 1 of field a2 of field ele3, a substructure of local structure substruct2

Reference field values individually in structure arrays

To reference the value of a field in a structure array, you must index into the array to the structure of interest and then reference that structure's field individually using dot notation, as in this example:

```
...
```

```
y = X(1).a % Extracts the value of field a
          % of the first structure in array X
...

```

To reference all the values of a particular field for each structure in an array, use this notation in a for loop, as in this example:

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,5);
for i = 1:5
    X(i).a = i;
    X(i).b = i+1;
end

```

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 8-7 for more information.

Do not reference fields dynamically

You cannot reference fields in a structure by using dynamic names, which express the field as a variable expression that MATLAB evaluates at run time (see “Generate Field Names from Variables”).

Assign Values to Structures and Fields

Use these guidelines when assigning values to a structure, substructure, or field for code generation:

Field properties must be consistent across structure-to-structure assignments

If:	Then:
Assigning one structure to another structure.	Define each structure with the same number, type, and size of fields.
Assigning one structure to a substructure of a different structure and vice versa.	Define the structure with the same number, type, and size of fields as the substructure.
Assigning an element of one structure to an element of another structure.	The elements must have the same type and size.

Do not use field values as constants

The values stored in the fields of a structure are not treated as constant values in generated code. Therefore, you cannot use field values to set the size or class of other data. For example, the following code generates a compiler error:

```
...
Y.a = 3;
X = zeros(Y.a); % Generates an error
```

In this example, even though you set field `a` of structure `Y` to the value `3`, `Y.a` is not a constant in generated code and, therefore, it is not a valid argument to pass to the function `zeros`.

Do not assign mxArray to structures

You cannot assign `mxArrays` to structure elements; convert `mxArrays` to known types before code generation (see “Working with `mxArrays`” on page 13-17).

Pass Large Structures as Input Parameters

If you generate a MEX function for a MATLAB function that takes a large structure as an input parameter, for example, a structure containing fields that are matrices, the MEX function might fail to load. This load failure occurs because, when you generate a MEX function from a MATLAB function that has input parameters, the code generation software allocates memory for these input parameters on the stack. To avoid this issue, pass the structure by reference to the MATLAB function. For example, if the original function signature is:

```
y = foo(a, S)
```

where S is the structure input, rewrite the function to:

```
[y, S] = foo(a, S)
```


Code Generation for Enumerated Data

- “Enumerated Data Definition for Code Generation” on page 9-2
- “Enumerated Types Supported for Code Generation” on page 9-3
- “When to Use Enumerated Data for Code Generation” on page 9-5
- “Generate Code for Enumerated Data from MATLAB Algorithms” on page 9-6
- “Define Enumerated Data for Code Generation” on page 9-8
- “Instantiate Enumerated Types for Code Generation” on page 9-10
- “Operations on Enumerated Data Allowed for Code Generation” on page 9-11
- “Include Enumerated Data in Control Flow Statements” on page 9-14
- “Customize Enumerated Types Based on int32” on page 9-20
- “Control Names of Enumerated Type Values in Generated Code” on page 9-26
- “Change and Reload Enumerated Data Types” on page 9-28
- “Restrictions on Use of Enumerated Data in for-Loops” on page 9-29
- “Toolbox Functions That Support Enumerated Types for Code Generation” on page 9-30

Enumerated Data Definition for Code Generation

To generate efficient standalone code for enumerated data, you must define and use enumerated types differently than you normally would when running your code in the MATLAB environment:

What's Different	More Information
Supports integer-based enumerated types only	"Enumerated Types Supported for Code Generation" on page 9-3
Name of each enumerated data type must be unique	"Naming Enumerated Types for Code Generation" on page 9-9
Each enumerated data type must be defined in a separate file on the MATLAB path	"Define Enumerated Data for Code Generation" on page 9-8 and "How to Generate Code for Enumerated Data" on page 9-6
Restricted set of operations	"Operations on Enumerated Data Allowed for Code Generation" on page 9-11
Restricted use in for-loops	"Restrictions on Use of Enumerated Data in for-Loops" on page 9-29

Enumerated Types Supported for Code Generation

Enumerated Type Based on int32

This enumerated data type is based on the built-in type `int32`. Use this enumerated type when generating code from MATLAB algorithms.

Syntax

```
classdef(Enumeration) type_name < int32
```

Example

```
classdef(Enumeration) PrimaryColors < int32
    enumeration
        Red(1),
        Blue(2),
        Yellow(4)
    end
end
```

In this example, the statement `classdef(Enumeration) PrimaryColors < int32` means that the enumerated type `PrimaryColors` is based on the built-in type `int32`. As such, `PrimaryColors` inherits the characteristics of the `int32` type, as well as defining its own unique characteristics. For example, `PrimaryColors` is restricted to three enumerated values:

Enumerated Value	Enumerated Name	Underlying Numeric Value
Red(1)	Red	1
Blue(2)	Blue	2
Yellow(4)	Yellow	4

How to Use

Define enumerated data in MATLAB code and compile the source file. For example, to generate C/C++ code from your MATLAB source, you can use

`codegen`, as described in “Generate Code for Enumerated Data from MATLAB Algorithms” on page 9-6.

Note `codegen` requires a MATLAB Coder license.

When to Use Enumerated Data for Code Generation

You can use enumerated types to represent program states and to control program logic, especially when you need to restrict data to a finite set of values and refer to these values by name. Even though you can sometimes achieve these goals by using integers or strings, enumerated types offer the following advantages:

- Provide more readable code than integers
- Allow more robust error checking than integers or strings

For example, if you mistype the name of an element in the enumerated type, you get a compile-time error that the element does not belong to the set of allowable values.

- Produce more efficient code than strings

For example, comparisons of enumerated values execute faster than comparisons of strings.

Generate Code for Enumerated Data from MATLAB Algorithms

Step	Action	How?
1	Define an enumerated data type that inherits from <code>int32</code> .	See “Define Enumerated Data for Code Generation” on page 9-8.
2	Instantiate the enumerated type in your MATLAB algorithm.	See “Instantiate Enumerated Types for Code Generation” on page 9-10.
3	Compile the function with <code>codegen</code> .	See “How to Generate Code for Enumerated Data” on page 9-6.

This workflow requires a MATLAB Coder license.

How to Generate Code for Enumerated Data

Use the command `codegen` to generate MEX, C, or C++ code from the MATLAB algorithm that contains the enumerated data (requires a MATLAB Coder license). Each enumerated data type must be defined on the MATLAB path in a separate file as a class derived from the built-in type `int32`. See “Define Enumerated Data for Code Generation” on page 9-8.

If your function has inputs, you must specify the properties of these inputs to `codegen`. For an enumerated data input, use the `-args` option to pass one of its allowable values as a sample value. For example, the following `codegen` command specifies that the function `displayState` takes one input of enumerated data type `sysMode`.

```
codegen displayState -args {sysMode.ON}
```

After executing this command, `codegen` generates a platform-specific MEX function that you can test in MATLAB. For example, to test `displayState`, type the following command:

```
displayState(sysMode.OFF)
```

You should get the following result:

```
ans =
```

```
    RED
```

Define Enumerated Data for Code Generation

Follow these steps to define enumerated data for code generation from MATLAB algorithms:

- 1 Create a class definition file.

In the MATLAB Command Window, select **File > New > Class**.

- 2 Enter the class definition as follows:

```
classdef(Enumeration) EnumTypeName < int32
```

For example, the following code defines an enumerated type called `sysMode`:

```
classdef(Enumeration) sysMode < int32
    ...
end
```

EnumTypeName is a case-sensitive string that must be unique among data type names and workspace variable names. It must inherit from the built-in type `int32`.

- 3 Define enumerated values in an enumeration section as follows:

```
classdef(Enumeration) EnumTypeName < int32
    enumeration
        EnumName(N)
        ...
    end
end
```

For example, the following code defines a set of two values for enumerated type `sysMode`:

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0)
        ON(1)
    end
end
```

Each enumerated value consists of a string *EnumName* and an underlying integer *N*. Each *EnumName* must be unique within its type, but can also appear in other enumerated types. The underlying integers need not be either consecutive or ordered, nor must they be unique within the type or across types.

4 Save the file on the MATLAB path.

The name of the file must match the name of the enumerated data type. The match is case sensitive.

To add a folder to the MATLAB search path, type `addpath pathname` at the MATLAB command prompt. For more information, see “What Is the MATLAB Search Path?”, `addpath`, and `savepath`.

For examples of enumerated data type definitions, see “Define Enumerated Data for Code Generation” on page 9-8.

Naming Enumerated Types for Code Generation

You must use a unique name for each enumerated data type. The name of an enumerated data type cannot match the name of a toolbox function supported for code generation, or another data type or a variable in the MATLAB base workspace. Otherwise, a name conflict occurs.

For example, you cannot name an enumerated data type `mode` because MATLAB for code generation provides a toolbox function of the same name.

For a list of toolbox functions supported for code generation, see “Functions Supported for Code Generation — Alphabetical List” on page 4-2.

Instantiate Enumerated Types for Code Generation

To instantiate an enumerated type for code generation from MATLAB algorithms, use dot notation to specify *ClassName.EnumName*. For an example, see “Include Enumerated Data in Control Flow Statements” on page 9-14.

Operations on Enumerated Data Allowed for Code Generation

To generate efficient standalone code for enumerated data, you are restricted to the following operations. The examples are based on the definitions of the enumeration type `LEDcolor` described in “Class Definition: `LEDcolor`” on page 9-14.

Assignment Operator, `=`

Example	Result
<pre>xon = LEDcolor.GREEN xoff = LEDcolor.RED</pre>	<pre>xon = GREEN xoff = RED</pre>

Relational Operators, `<` `>` `<=` `>=` `==` `~=`

Example	Result
<pre>xon == xoff</pre>	<pre>ans = 0</pre>
<pre>xon <= xoff</pre>	<pre>ans = 1</pre>
<pre>xon > xoff</pre>	<pre>ans = 0</pre>

Cast Operation

Example	Result
<code>double(LEDcolor.RED)</code>	ans = 2
<code>z = 2</code> <code>y = LEDcolor(z)</code>	z = 2 y = RED

Indexing Operation

Example	Result
<code>m = [1 2]</code> <code>n = LEDcolor(m)</code> <code>p = n(LEDcolor.GREEN)</code>	m = 1 2 n = GREEN RED p = GREEN

Control Flow Statements: if, switch, while

Statement	Example	Executable Example
if	<pre> if state == sysMode.ON led = LEDcolor.GREEN; else led = LEDcolor.RED; end </pre>	<p>“if Statement with Enumerated Data Types” on page 9-14</p>
switch	<pre> switch button case VCRButton.Stop state = VCRState.Stop; case VCRButton.PlayOrPause state = VCRState.Play; case VCRButton.Next state = VCRState.Forward; case VCRButton.Previous state = VCRState.Rewind; otherwise state = VCRState.Stop; end </pre>	<p>“switch Statement with Enumerated Data Types” on page 9-15</p>
while	<pre> while state ~= State.Ready switch state case State.Standby initialize(); state = State.Boot; case State.Boot boot(); state = State.Ready; end end end </pre>	<p>“while Statement with Enumerated Data Types” on page 9-18</p>

Include Enumerated Data in Control Flow Statements

The following control statements work with enumerated operands in generated code. However, there are restrictions (see “Restrictions on Use of Enumerated Data in for-Loops” on page 9-29).

if Statement with Enumerated Data Types

This example is based on the definition of the enumeration types `LEDcolor` and `sysMode`. The function `displayState` uses these enumerated data types to activate an LED display.

Class Definition: `sysMode`

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0)
        ON(1)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `sysMode.m`.

Class Definition: `LEDcolor`

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
end
```

This definition must reside on the MATLAB path in a file called `LEDcolor.m`.

MATLAB Function: `displayState`

This function uses enumerated data to activate an LED display, based on the state of a device. It lights a green LED display to indicate the ON state and lights a red LED display to indicate the OFF state.

```
function led = displayState(state)
%#codegen

if state == sysMode.ON
    led = LEDcolor.GREEN;
else
    led = LEDcolor.RED;
end
```

Build and Test a MEX Function for displayState

- 1 Generate a MEX function for displayState. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen displayState -args {sysMode.ON}
```

- 2 Test the function. For example,

```
displayState(sysMode.OFF)
```

You should get the following result:

```
ans =

    RED
```

switch Statement with Enumerated Data Types

This example is based on the definition of the enumeration types `VCRState` and `VCRButton`. The function `VCR` uses these enumerated data types to set the state of the VCR.

Class Definition: VCRState

```
classdef(Enumeration) VCRState < int32
    enumeration
        Stop(0),
        Pause(1),
        Play(2),
        Forward(3),
```

```
        Rewind(4)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRState.m`.

Class Definition: VCRButton

```
classdef(Enumeration) VCRButton < int32
    enumeration
        Stop(1),
        PlayOrPause(2),
        Next(3),
        Previous(4)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRButton.m`.

MATLAB Function: VCR

This function uses enumerated data to set the state of a VCR, based on the initial state of the VCR and the state of the VCR button.

```
function s = VCR(button)
    %#codegen

    persistent state

    if isempty(state)
        state = VCRState.Stop;
    end

    switch state
        case {VCRState.Stop, VCRState.Forward, VCRState.Rewind}
            state = handleDefault(button);
        case VCRState.Play
            switch button
```

```

        case VCRButton.PlayOrPause, state = VCRState.Pause;
        otherwise, state = handleDefault(button);
    end
case VCRState.Pause
    switch button
        case VCRButton.PlayOrPause, state = VCRState.Play;
        otherwise, state = handleDefault(button);
    end
end
s = state;

function state = handleDefault(button)
switch button
    case VCRButton.Stop, state = VCRState.Stop;
    case VCRButton.PlayOrPause, state = VCRState.Play;
    case VCRButton.Next, state = VCRState.Forward;
    case VCRButton.Previous, state = VCRState.Rewind;
    otherwise, state = VCRState.Stop;
end

```

Build and Test a MEX Function for VCR

- 1 Generate a MEX function for VCR. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen -args {VCRButton.Stop} VCR
```

- 2 Test the function. For example,

```
s = VCR(VCRButton.Stop)
```

You should get the following result:

```
s =

    Stop
```

while Statement with Enumerated Data Types

This example is based on the definition of the enumeration type `State`. The function `Setup` uses this enumerated data type to set the state of a device.

Class Definition: `State`

```
classdef(Enumeration) State < int32
    enumeration
        Standby(0),
        Boot(1),
        Ready(2)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `State.m`.

MATLAB Function: `Setup`

The following function `Setup` uses enumerated data to set the state of a device.

```
function s = Setup(initState)
%#codegen

state = initState;

if isempty(state)
    state = State.Standby;
end

while state ~= State.Ready
    switch state
        case State.Standby
            initialize();
            state = State.Boot;
        case State.Boot
            boot();
            state = State.Ready;
    end
end
```



```
s = state;

function initialize()
% Perform initialization.

function boot()
% Boot the device.
```

Build and Test a MEX Executable for Setup

- 1 Generate a MEX executable for Setup. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen Setup -args {State.Standby}
```

- 2 Test the function. For example,

```
s = Setup(State.Standby)
```

You should get the following result:

```
s =

    Ready
```

Customize Enumerated Types Based on int32

About Customizing Enumerated Types

You can customize an enumerated type by using the same techniques that work with MATLAB classes, as described in *Modifying Superclass Methods and Properties*. A primary source of customization are the methods associated with an enumerated type.

Enumerated class definitions can include an optional methods section. You can override the following methods to customize the behavior of an enumerated type. To override a method, include a customized version of the method in the methods section in the enumerated class definition. If you do not want to override the inherited methods, omit the methods section.

Method	Description	Default Value Returned or Specified	When to Use
addClassNameToEnumNames	Specifies whether the class name becomes a prefix in the generated code.	true — prefix is used	If you do not want the class name to become a prefix in the generated code, override this method to set the return value to <code>false</code> . See “Control Names of Enumerated Type Values in Generated Code” on page 9-26.
getDefaultValue	Returns the default enumerated value.	''	If you want the default value for the enumerated type to be something other than the first value listed in the enumerated class definition, override this method to specify a default value. See “Specify a Default Enumerated Value” on page 9-22.

Method	Description	Default Value Returned or Specified	When to Use
getHeaderFile	Specifies the file in which the enumerated class is defined for code generation.	' '	If you want to use an enumerated class definition that is specified in a custom header file, override this method to return the path to this header file. In this case, the code generation software does not generate the class definition. See “Specify a Header File” on page 9-23

Specify a Default Enumerated Value

The code generation software and related generated code use an enumerated data type’s default value when you do not provide an initial value.

Unless you specify otherwise, the default value for an enumerated type is the first value in the enumerated class definition. To specify a different default value, add your own `getDefaultValue` method to the methods section. The following code shows a shell for the `getDefaultValue` method:

```
function retVal = getDefaultValue()
% GETDEFAULTVALUE Returns the default enumerated value.
% This value must be an instance of the enumerated class.
% If this method is not defined, the first enumerated value is used.
    retVal = ThisClass.EnumName;
end
```

To customize this method, provide a value for `ThisClass.EnumName` that specifies the desired default. `ThisClass` must be the name of the class within

which the method exists. EnumName must be the name of an enumerated value defined in that class. For example:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
    methods (Static)
        function y = getDefaultvalue()
            y = LEDcolor.RED;
        end
    end
end
```

This example defines the default as LEDcolor.RED. If this method does not appear, the default value would be LEDcolor.GREEN, because that is the first value listed in the enumerated class definition.

Specify a Header File

To prevent the declaration of an enumerated type from being embedded in the generated code, allowing you to provide the declaration in an external file, include the following method in the enumerated class's methods section:

```
function y = getHeaderFile()
% GETHEADERFILE File where type is defined for generated code.
% If specified, this file is #included where required in the code.
% Otherwise, the type is written out in the generated code.
y = 'filename';
end
```

Substitute a legal filename for filename. Be sure to provide a filename suffix, typically .h. Providing the method replaces the declaration that would otherwise have appeared in the generated code with a #include statement like:

```
#include "imported_enum_type.h"
```

The getHeaderFile method does not create the declaration file itself. You must provide a file of the specified name that declares the enumerated data

type. The file can also contain definitions of enumerated types that you do not use in your MATLAB code.

For example, to use the definition of LEDcolor in `my_LEDcolor.h`:

- 1 Modify the definition of LEDcolor to override the `getHeaderFile` method to return the name of the external header file:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end

    methods(Static)
        function y=getHeaderFile()
            y='my_LEDcolor.h';
        end
    end
end
```

- 2 In the current folder, provide a header file, `my_LEDcolor.h`, that contains the definition:

```
typedef enum LEDcolor
{
    GREEN = 1,
    RED
} LEDcolor;
```

- 3 Generate a library for the function `displayState` that takes one input of enumerated data type `sysMode`.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

`codegen` generates a C static library with the default name, `displayState`, and supporting files in the default folder, `codegen/lib/displayState`.

- 4 Click the *View Report* link.

- 5** In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The header file contains a `#include` statement for the external header file.

```
#include "my_LEDcolor.h"
```

It does not include a declaration for the enumerated class.

Control Names of Enumerated Type Values in Generated Code

This example shows how to control the name of enumerated type values in code generated by MATLAB Coder. (Requires a MATLAB Coder license.) The example uses the enumerated data type definitions and function `displayState` described in “Include Enumerated Data in Control Flow Statements” on page 9-14.

- 1 Generate a library for the function `displayState` that takes one input of enumerated data type `sysMode`.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

`codegen` generates a C static library with the default name, `displayState`, and supporting files in the default folder, `codegen/lib/displayState`.

- 2 Click the *View Report* link.
- 3 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The report displays the header file containing the enumerated data type definition.

```
typedef enum LEDcolor
{
    LEDcolor_GREEN = 1,
    LEDcolor_RED
} LEDcolor;
```

The enumerated value names include the class name prefix `LEDcolor_`.

- 4 Modify the definition of `LEDcolor` to override the `addClassNameToEnumNames` method. Set the return value to `false` instead of `true` so that the enumerated value names in the generated code do not contain the class prefix.

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
```



```

        RED(2),
    end

    methods(Static)
        function y=addClassNameToEnumNames()
            y=false;
        end
    end
end
end

```

5 Clear existing class instances:

```
clear classes
```

6 Generate code again.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

7 Open the code generation report and look at the enumerated type definition in `displayState_types.h`.

```
typedef enum LEDcolor
{
    GREEN = 1,
    RED
} LEDcolor;
```

This time the enumerated value names do not include the class name prefix.

For more information, see:

- `codegen`
- “Include Enumerated Data in Control Flow Statements” on page 9-14 for a description of the example function `displayState` and its enumerated type definitions

Change and Reload Enumerated Data Types

You can change the definition of an enumerated data type by editing and saving the file that contains the definition. You do not need to inform MATLAB that a class definition has changed. MATLAB automatically reads the modified definition when you save the file. However, the class definition changes do not take full effect if class instances (enumerated values) exist that reflect the previous class definition. Such instances might exist in the base workspace or might be cached. The following table explains options for removing instances of an enumerated data type from the base workspace and cache.

If In Base Workspace...	If In Cache...
Do one of the following: <ul style="list-style-type: none">• Locate and delete specific obsolete instances.• Delete the classes from the workspace by using the <code>clear classes</code> command. For more information, see <code>clear</code>.	<ul style="list-style-type: none">• Clear MEX functions that are caching instances of the class.

Restrictions on Use of Enumerated Data in for-Loops

Do not use enumerated data as the loop counter variable in for-loops

To iterate over a range of enumerated data with consecutive values, you can cast the enumerated data to `int32` in the loop counter.

For example, suppose you define an enumerated type `ColorCodes` as follows:

```
classdef(Enumeration) ColorCodes < int32
    enumeration
        Red(1),
        Blue(2),
        Green(3)
        Yellow(4)
        Purple(5)
    end
end
```

Because the enumerated values are consecutive, you can use `ColorCodes` data in a for-loop like this:

```
...
for i = int32(ColorCodes.Red):int32(ColorCodes.Purple)
    c = ColorCodes(i);
    ...
end
```

Toolbox Functions That Support Enumerated Types for Code Generation

The following MATLAB toolbox functions support enumerated types for code generation:

- `cast`
- `cat`
- `circshift`
- `flipdim`
- `fliplr`
- `flipud`
- `histc`
- `ipermute`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `shiftdim`
- `sort`
- `sortrows`

- squeeze

Code Generation for MATLAB Classes

- “MATLAB Classes Definition for Code Generation” on page 10-2
- “Classes That Support Code Generation” on page 10-8
- “Generate Code for MATLAB Value Classes” on page 10-9
- “Generate Code for MATLAB Handle Classes and System Objects” on page 10-15
- “MATLAB Classes in Code Generation Reports” on page 10-17
- “Troubleshooting Issues with MATLAB Classes” on page 10-20

MATLAB Classes Definition for Code Generation

To generate efficient standalone code for MATLAB classes, you must use classes differently than you normally would when running your code in the MATLAB environment.

What's Different	More Information
Class must be in a single file. Because of this limitation, code generation is not supported for a class definition that uses an @-folder.	“Creating a Single, Self-Contained Class Definition File”
Restricted set of language features.	“Language Limitations” on page 10-2
Restricted set of code generation features.	“Code Generation Features Not Compatible with Classes” on page 10-3
Definition of class properties.	“Defining Class Properties for Code Generation” on page 10-4
Use of handle classes.	“Generate Code for MATLAB Handle Classes and System Objects” on page 10-15
Calls to base class constructor.	“Calls to Base Class Constructor” on page 10-6
Global variables containing MATLAB objects are not supported for code generation.	N/A

Language Limitations

Although code generation support is provided for common features of classes such as properties and methods, there are a number of advanced features which are not supported, such as:

- Events
- Listeners
- Arrays of objects

- Recursive data structures
 - Linked lists
 - Trees
 - Graphs
- Overloadable operators `subsref`, `subsassign`, and `subsindex`

In MATLAB, classes can define their own versions of the `subsref`, `subsassign`, and `subsindex` methods. Code generation does not support classes that have their own definitions of these methods.
- The empty method

In MATLAB, classes have a built-in static method, `empty`, which creates an empty array of the class. Code generation does not support this method.
- The following MATLAB handle class methods:
 - `addlistener`
 - `delete`
 - `eq`
 - `findobj`
 - `findpro`

Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

```
codegen ClassNameA
```
- If an entry-point MATLAB function has an input or output that is a MATLAB class, you cannot generate code for this function.

For example, if function `f00` takes one input, `a`, that is a MATLAB object, you cannot generate code for `f00` by executing:

```
codegen foo -args {a}
```

- You cannot generate code for a value class that has a `set.prop` method. For example, you cannot generate code for the following `Square` class because of the `set.side` method.

```
classdef Square < Shape %#codegen
    properties
        side;
    end
    methods
        function obj = Square(side)
            obj = obj@Shape(side^2);
            obj.side = side;
        end
        function set.side(obj,value)
            obj.side = value;
            obj.area = value^2;
        end
    end
end
```

To generate code for this class, modify the class definition to remove the `set.side` method.

- Code generation does not support assigning an object of a value class into a nontunable property. For example, `obj.prop=v;` is invalid when `prop` is a nontunable property and `v` is an object based on a value class.
- You cannot use `coder.extrinsic` to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to the `coder.ceval` function.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.
- The `coder.nullcopy` function does not support MATLAB classes as inputs.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you normally would when running your code in the MATLAB environment:

- After defining a property, do not assign it an incompatible type. Do not use a property before attempting to grow it.

When you define class properties for code generation, consider the same factors that you take into account when defining variables. In the MATLAB language, variables can change their class, size, or complexity dynamically at run time so you can use the same variable to hold a value of varying class, size, or complexity. C and C++ use static typing. Before using variables, to determine their type, the code generation software requires a complete assignment to each variable. Similarly, before using properties, you must explicitly define their class, size, and complexity.

- Initial values:
 - If the property does not have an explicit initial value, the code generation software assumes that it is undefined at the beginning of the constructor. The code generation software does not assign an empty matrix as the default.
 - If the property does not have an initial value and the code generation software cannot determine that the property is assigned prior to first use, the software generates a compilation error.
 - For System objects, if a nontunable property is a structure, you must completely assign the structure. You cannot do partial assignment using subscripting.

For example, for a nontunable property, you can use the following assignment:

```
mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
```

You cannot use the following partial assignments:

```
mySystemObject.nonTunableProperty.fieldA = a;  
mySystemObject.nonTunableProperty.fieldB = b;
```

- If dynamic memory allocation is enabled, code generation supports variable-size properties for handle classes. Without dynamic memory allocation, you cannot generate code for handle classes that have variable-size properties.
- `coder. varsize` is not supported for class properties.

- MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.target` in MATLAB class property initialization, `coder.target` is ''.

Calls to Base Class Constructor

If a class constructor contains a call to the constructor of the base class, the call to the base class constructor must come before `for`, `if`, `return`, `switch` or `while` statements.

For example, if you define a class B based on class A:

```
classdef B < A
    methods
        function obj = B(varargin)
            if nargin == 0
                a = 1;
                b = 2;
            elseif nargin == 1
                a = varargin{1};
                b = 1;
            elseif nargin == 2
                a = varargin{1};
                b = varargin{2};
            end
            obj = obj@A(a,b);
        end
    end
end
```

Because the class definition for B uses an `if` statement before calling the base class constructor for A, you cannot generate code for function `callB`:

```
function [y1,y2] = callB
x = B;
y1 = x.p1;
y2 = x.p2;
```

```
end
```

However, you can generate code for `callB` if you define class `B` as:

```
classdef B < A
    methods
        function obj = NewB(varargin)
            [a,b] = getaandb(varargin{:});
            obj = obj@A(a,b);
        end
    end
end
```

```
function [a,b] = getaandb(varargin)
if nargin == 0
    a = 1;
    b = 2;
elseif nargin == 1
    a = varargin{1};
    b = 1;
elseif nargin == 2
    a = varargin{1};
    b = varargin{2};
end
end
```

Classes That Support Code Generation

You can generate code for MATLAB value and handle classes and user-defined System objects. Your class can have multiple methods and properties and can inherit from multiple classes.

To generate code for:	Example:
Value classes	“Generate Code for MATLAB Value Classes” on page 10-9
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 10-15

For more information, see:

- “Classes in the MATLAB Language”
- “MATLAB Classes Definition for Code Generation” on page 10-2

Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

- 1 In a writable folder, create a MATLAB value class, Shape. Save the code as Shape.m.

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out = obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
    methods(Abstract = true)
        getarea(obj);
    end
    methods(Static)
        function d = distanceBetweenShapes(shape1,shape2)
            xDist = abs(shape1.centerX - shape2.centerX);
            yDist = abs(shape1.centerY - shape2.centerY);
            d = sqrt(xDist^2 + yDist^2);
        end
    end
end
```

- 2** In the same folder, create a class, `Square`, that is a subclass of `Shape`. Save the code as `Square.m`.

```
classdef Square < Shape
% Create a Square at coordinates center X and center Y
% with sides of length of side
    properties
        side;
    end
    methods
        function obj = Square(side,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.side = side;
        end
        function Area = getarea(obj)
            Area = obj.side^2;
        end
    end
end
```

- 3** In the same folder, create a class, `Rhombus`, that is a subclass of `Shape`. Save the code as `Rhombus.m`.

```
classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)
            Area = 0.5*obj.diag1*obj.diag2;
        end
    end
end
```

- 4** Write a function that uses this class.


```
function [TotalArea, Distance] = use_shape
%#codegen
s = Square(2,1,2);
r = Rhombus(3,4,7,10);
TotalArea = s.area + r.area;
Distance = Shape.distanceBetweenShapes(s,r);
```

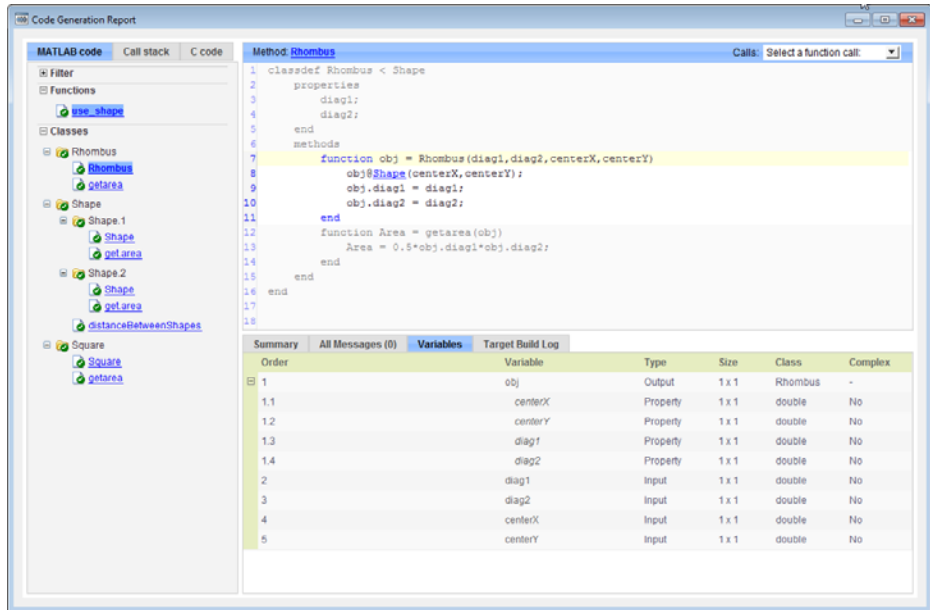
- 5 Generate a static library for `use_shape` and generate a code generation report.

```
codegen -config:lib -report use_shape
```

`codegen` generates a C static library with the default name, `use_shape`, and supporting files in the default folder, `codegen/lib/use_shape`.

- 6 Click the *View report* link.
- 7 In the report, on the **MATLAB code** tab, click the link to the Rhombus class.

The report displays the class definition of the Rhombus class and highlights the class constructor. On the **Variables** tab, it provides details of the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties. To view the complete list of properties, expand the list as shown for `obj`.



8 At the top right side of the report, expand the **Calls** list.

The **Calls** list shows that there is a call to the Rhombus constructor from use_shape and that this constructor calls the Shape constructor.

The screenshot displays the MATLAB Code Generation Report for the Rhombus class. The left sidebar shows a tree view of the code, with the Rhombus class expanded to show its constructor. The main window shows the MATLAB code for the Rhombus class, which inherits from the Shape class. The constructor function is highlighted, and a callout menu is open over the `obj@Shape` call, showing options to select a function call.

```

1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape(centerX,centerY);
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17
18

```

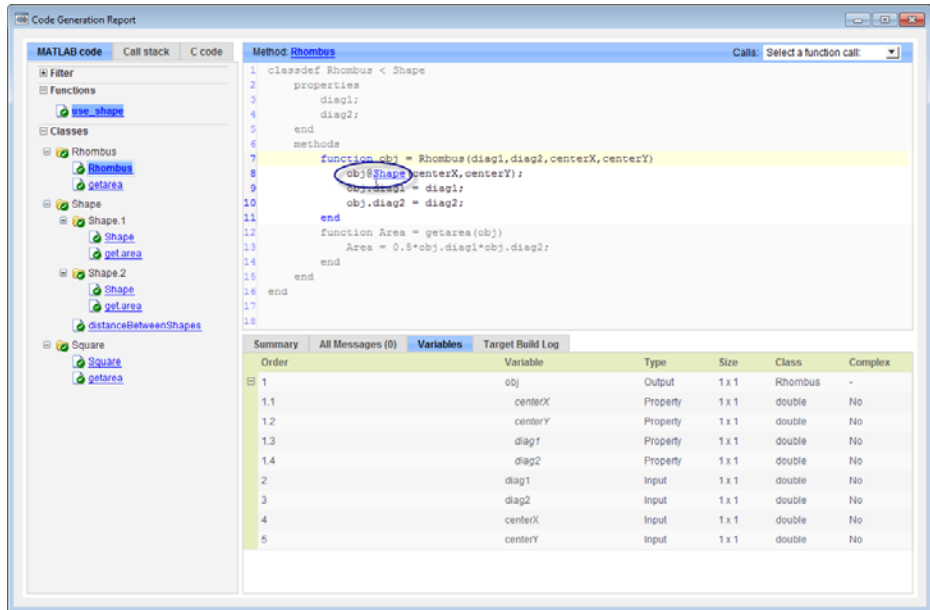
The callout menu shows the following options:

- Select a function call:
- Calls to this function:
- From use_shape line 4
- Calls from this function
- From line 8 to Shape

The bottom of the report shows a table with the following columns: Summary, All Messages (0), Variables, and Target Build Log. The Variables table is expanded to show the following variables:

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1	centerX	Property	1 x 1	double	No
1.2	centerY	Property	1 x 1	double	No
1.3	diag1	Property	1 x 1	double	No
1.4	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

- 9 The constructor for the Rhombus class calls the Shape method of the base Shape class: `obj@Shape`. In the report, click the Shape link in this call.



The link takes you to the Shape method in the Shape class definition.

Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report.

- 1 In a writable folder, create a System object, `AddOne`, which subclasses from `matlab.System`. Save the code as `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access=protected)
        % stepImpl method is called by the step method
        function y = stepImpl(~,x)
            y = x+1;
        end
    end
end
```

- 2 Write a function that uses this System object.

```
function y = testAddOne(x)
%#codegen
    p = AddOne();
    y = p.step(x);
end
```

- 3 Generate a MEX function for this code.

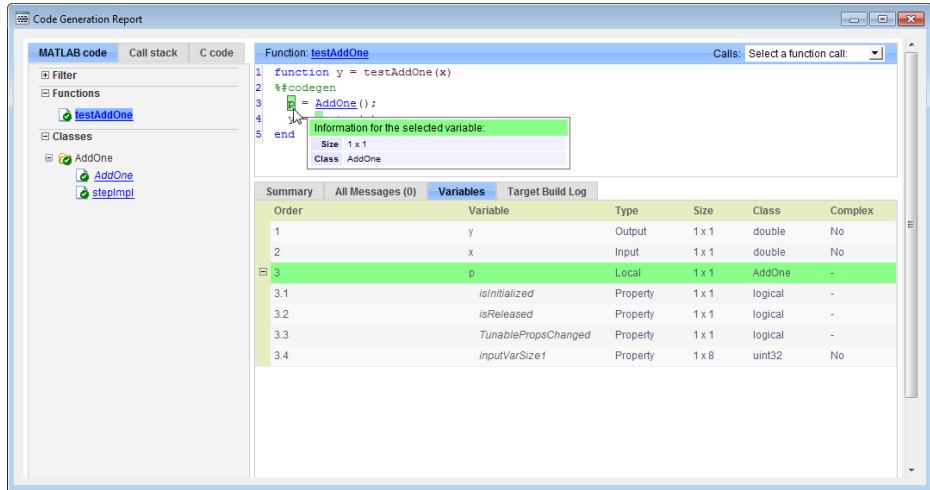
```
codegen -report testAddOne -args {0}
```

The `-report` option instructs `codegen` to generate a code generation report, even if no errors or warnings occur. The `-args` option specifies that the `testAddOne` function takes one scalar double input.

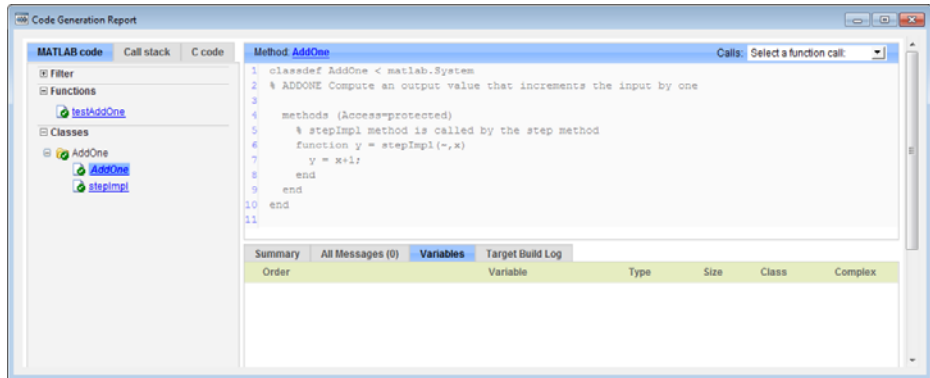
```
>> codegen -report testAddOne -args {0}
Code generation successful: View report
```

- 4 Click the *View report* link.

- In the report, on the **MATLAB Code** tab **Functions** panel, click `testAddOne`, then click the **Variables** tab. You can view information about the variable `p` on this tab.



- To view the class definition, on the **Classes** panel, click `AddOne`.



MATLAB Classes in Code Generation Reports

What Reports Tell You About Classes

Code generation reports:

- Provide a hierarchical tree of the classes used in your MATLAB code.
- Display a list of methods for each class in the MATLAB code tab.
- Display the objects used in your MATLAB code together with their properties on the **Variables** tab.
- Provide a filter so that you can sort methods by class, size, and complexity.
- List the set of calls from and to the selected method in the **Calls** list.

How Classes Appear in Code Generation Reports

In the MATLAB Code Tab

The report displays an alphabetical hierarchical list of the classes used in the your MATLAB code. For each class, you can:

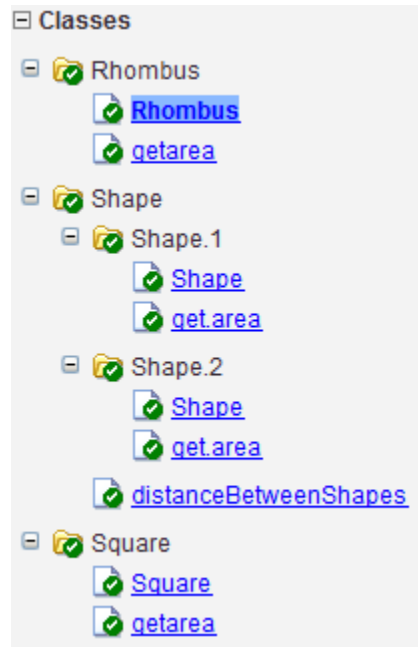
- Expand the class information to view the class methods.
- View a class method by clicking its name. The report displays the methods in the context of the full class definition.
- Filter the methods by size, complexity, and class by using the **Filter functions and methods** option.

Default Constructors. If a class has a default constructor, the report displays the constructor in italics.

Specializations. If the same class is specialized into multiple different classes, the report differentiates the specializations by grouping each one under a single node in the tree. The report associates the class definition functions and static methods with the primary node. It associates the instance-specific methods with the corresponding specialized node.

For example, consider a base class, `Shape` that has two specialized subclasses, `Rhombus` and `Square`. The `Shape` class has an abstract method, `getarea`,

and a static method, `distanceBetweenShapes`. The code generation report, displays a node for the specialized `Rhombus` and `Square` classes with their constructors and `getarea` method. It displays a node for the `Shape` class and its associated static method, `distanceBetweenShapes`, and two instances of the `Shape` class, `Shape1` and `Shape2`.



Packages. If you define classes as part of a package, the report displays the package in the list of classes. You can expand the package to view the classes that it contains. For more information about packages, see “Packages Create Namespaces”.

In the Variables Tab

The report displays the objects in the selected function or class. By default, for classes that have properties, the list of properties is collapsed. Click the + symbol next to the object name to open the list.

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable `obj1` that is a

MATLAB object with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

In the Call Stack

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

How to Generate a Code Generation Report

Add the `-report` option to your `codegen` command (requires a MATLAB Coder license)

Troubleshooting Issues with MATLAB Classes

Class *class* does not have a property with name *name*

If a MATLAB class has a method, `mymethod`, that returns a handle class with a property, `myprop`, you cannot generate code for the following type of assignment:

```
obj.mymethod().myprop=...
```

For example, consider the following classes:

```
classdef MyClass < handle
    properties
        myprop
    end
    methods
        function this = MyClass
            this.myprop = MyClass2;
        end
        function y = mymethod(this)
            y = this.myprop;
        end
    end
end
```

```
classdef MyClass2 < handle
    properties
        aa
    end
end
```

You cannot generate code for function `foo`.

```
function foo

h = MyClass;

h.mymethod().aa = 12;
```

In this function, `h.mymethod()` returns a handle object of type `MyClass2`. In MATLAB, the assignment `h.mymethod().aa = 12;` changes the property of that object. Code generation does not support this assignment.

Workaround

Rewrite the code to return the object and then assign a value to a property of the object.

```
function foo

h = MyClass;

b=h.mymethod();
b.aa=12;
```


Code Generation for Function Handles

- “Function Handle Definition for Code Generation” on page 11-2
- “Define and Pass Function Handles for Code Generation” on page 11-3
- “Function Handle Limitations for Code Generation” on page 11-5

Function Handle Definition for Code Generation

You can use function handles to invoke functions indirectly and parameterize operations that you repeat frequently. You can perform the following operations with function handles:

- Define handles that reference user-defined functions and built-in functions supported for code generation (see “Functions Supported for Code Generation — Alphabetical List” on page 4-2)

Note You cannot define handles that reference extrinsic MATLAB functions.

- Define function handles as scalar values
- Pass function handles as arguments to other functions (excluding extrinsic functions)

To generate efficient standalone code for enumerated data, you are restricted to using a subset of the operations you can perform with function handles in MATLAB, as described in “Function Handle Limitations for Code Generation” on page 11-5

Define and Pass Function Handles for Code Generation

The following code example shows how to define and call function handles for code generation. You can copy the example to a MATLAB Function block in Simulink or MATLAB function in Stateflow. To convert this function to a MEX function using `codegen`, uncomment the two calls to the `assert` function, highlighted below:

```
function addval(m)
%#codegen

% Define class and size of primary input m
% Uncomment next two lines to build MEX function with codegen
% assert(isa(m,'double'));
% assert(all (size(m) == [3 3]));

% Pass function handle to addone
% to add one to each element of m
m = map(@addone, m);
disp(m);

% Pass function handle to addtwo
% to add two to each element of m
m = map(@addtwo, m);
disp(m);

function y = map(f,m)
    y = m;
    for i = 1:numel(y)
        y(i) = f(y(i));
    end

function y = addone(u)
y = u + 1;

function y = addtwo(u)
y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed

by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

If you have MATLAB Coder, you can use the function `codegen` to convert the function `addval` to a MEX executable that you can run in MATLAB. Follow these steps:

- 1** At the MATLAB command prompt, issue this command:

```
codegen addval
```

- 2** Define and initialize a 3-by-3 matrix by typing a command like this at the MATLAB prompt:

```
m = zeros(3)
```

- 3** Execute the function by typing this command:

```
addval(m)
```

You should see the following result:

```
0    0    0
0    0    0
0    0    0

1    1    1
1    1    1
1    1    1

3    3    3
3    3    3
3    3    3
```

For more information, see “MEX Function Generation at the Command Line”.

Function Handle Limitations for Code Generation

Function handles must be scalar values.

You cannot store function handles in matrices or structures.

You cannot use the same bound variable to reference different function handles.

After you bind a variable to a specific function, you cannot use the same variable to reference two different function handles, as in this example

```
%Incorrect code
...
x = @plus;
x = @minus;
...
```

This code produces a compilation error.

You cannot pass function handles to or from extrinsic functions.

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions. For more information, see “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12

You cannot pass function handles to or from primary functions.

You cannot pass function handles as inputs to or outputs from primary functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle, 'function_handle') && isa(data, 'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as primary inputs. `plotFcn` attempts to call the function referenced by

the `fhandle` with the input data and plot the results. However, this code generates a compilation error, indicating that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of primary inputs.

You cannot view function handles from the debugger

You cannot display or watch function handles from the debugger. They appear as empty matrices.

Defining Functions for Code Generation

- “Specify Variable Numbers of Arguments” on page 12-2
- “Supported Index Expressions” on page 12-3
- “Apply Operations to a Variable Number of Arguments” on page 12-4
- “Implement Wrapper Functions” on page 12-7
- “Pass Property/Value Pairs” on page 12-8
- “Variable Length Argument Lists for Code Generation” on page 12-10

Specify Variable Numbers of Arguments

You can use `varargin` and `varargout` for passing and returning variable numbers of parameters to MATLAB functions called from a top-level function.

Common applications of `varargin` and `varargout` for code generation are to:

- “Apply Operations to a Variable Number of Arguments” on page 12-4
- “Implement Wrapper Functions” on page 12-7
- “Pass Property/Value Pairs” on page 12-8

Code generation relies on loop unrolling to produce simple and efficient code for `varargin` and `varargout`. This technique permits most common uses of `varargin` and `varargout`, but some uses are not allowed (see “Variable Length Argument Lists for Code Generation” on page 12-10). This following sections explain how to code effectively using these constructs.

For more information about using `varargin` and `varargout` in MATLAB functions, see [Passing Variable Numbers of Arguments](#).

Supported Index Expressions

In MATLAB, `varargin` and `varargout` are cell arrays. Generated code does not support cell arrays, but does allow you to use the most common syntax — curly braces `{}` — for indexing into `varargin` and `varargout` arrays, as in this example:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i};
end

```

You can use the following index expressions. The *exp* arguments must be constant expressions or depend on a loop index variable.

Expression		Description
<code>varargin</code> (read only)	<code>varargin{exp}</code>	Read the value of element <i>exp</i>
	<code>varargin{exp1: exp2}</code>	Read the values of elements <i>exp1</i> through <i>exp2</i>
	<code>varargin{:}</code>	Read the values of all elements
<code>varargout</code> (read and write)	<code>varargout{exp}</code>	Read or write the value of element <i>exp</i>

Note The use of `()` is not supported for indexing into `varargin` and `varargout` arrays.

Apply Operations to a Variable Number of Arguments

You can use `varargin` and `varargout` in `for`-loops to apply operations to a variable number of arguments. To index into `varargin` and `varargout` arrays in generated code, the value of the loop index variable must be known at compile time. Therefore, during code generation, the compiler attempts to automatically unroll these `for`-loops. Unrolling eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant. For example, the following function automatically unrolls its `for`-loop in the generated code:

```
##codegen
function [cmlen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmlen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

When to Force Loop Unrolling

To automatically unroll `for`-loops containing `varargin` and `varargout` expressions, the relationship between the loop index expression and the index variable must be determined at compile time.

In the following example, the function `fcn` cannot detect a logical relationship between the index expression `j` and the index variable `i`:

```
##codegen
function [x,y,z] = fcn(a,b,c)

[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;
for i = 1:length(varargin)
    j = j+1;
    varargout{j} = varargin{j};
```

```
end
```

As a result, the function does not unroll the loop and generates a compilation error:

```
Nonconstant expression or empty matrix.
This expression must be constant because
its value determines the size or class of some expression.
```

To fix the problem, you can force loop unrolling by wrapping the loop header in the function `coder.unroll`, as follows:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
    [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
    j = 0;
    for i = coder.unroll(1:length(varargin))
        j = j + 1;
        varargout{j} = varargin{j};
    end;

```

Using Variable Numbers of Arguments in a for-Loop

The following example multiplies a variable number of input dimensions in inches by 2.54 to convert them to centimeters:

```

%#codegen
function [cmlen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmlen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end

```

Key Points About the Example

- `varargin` and `varargout` appear in the local function `inch_2_cm`, not in the top-level function `conv_2_metric`.
- The index into `varargin` and `varargout` is a for-loop variable

For more information, see “Variable Length Argument Lists for Code Generation” on page 12-10.

Implement Wrapper Functions

You can use `varargin` and `varargout` to write wrapper functions that accept up to 64 inputs and pass them directly to another function.

Passing Variable Numbers of Arguments from One Function to Another

The following example passes a variable number of inputs to different optimization functions, based on a specified input method:

```
%#codegen
function answer = fcn(method,a,b,c)
answer = optimize(method,a,b,c);

function answer = optimize(method,varargin)
    if strcmp(method,'simple')
        answer = simple_optimization(varargin{:});
    else
        answer = complex_optimization(varargin{:});
    end
    ...
```

Key Points About the Example

- You can use `{:}` to read all elements of `varargin` and pass them to another function.
- You can mix variable and fixed numbers of arguments.

For more information, see “Variable Length Argument Lists for Code Generation” on page 12-10.

Pass Property/Value Pairs

You can use `varargin` to pass property/value pairs in functions. However, for code generation, you must take precautions to avoid type mismatch errors when evaluating `varargin` array elements in a `for`-loop:

If	Do This:
You assign <code>varargin</code> array elements to local variables in the <code>for</code> -loop	Verify that for all pairs, the size, type, and complexity are the same for each property and the same for each value
Properties or values have different sizes, types, or complexity	Do not assign <code>varargin</code> array elements to local variables in a <code>for</code> -loop; reference the elements directly

For example, in the following function `test1`, the sizes of the property strings and numeric values are not the same in each pair:

```

%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        name = varargin{i};
        value = varargin{i+1};
        switch name
            case 'size'
                v = set_size(v, value);
            case 'rgb'
                v = set_color(v, value);
            otherwise
            end
        end
    end
end

```

...

Generated code determines the size, type, and complexity of a local variable based on its first assignment. In this example, the first assignments occur in the first iteration of the for-loop:

- Defines local variable `name` with size equal to 4
- Defines local variable `value` with a size of scalar

However, in the second iteration, the size of the property string changes to 3 and the size of the numeric value changes to a vector, resulting in a type mismatch error. To avoid such errors, reference `varargin` array values directly, not through local variables, as highlighted in this code:

```

%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        switch varargin{i}
            case 'size'
                v = set_size(v, varargin{i+1});
            case 'rgb'
                v = set_color(v, varargin{i+1});
            otherwise
        end
    end
end
...

```

Variable Length Argument Lists for Code Generation

Do not use varargin or varargout in top-level functions

You **cannot** use varargin or varargout as arguments to top-level functions. A *top-level function* is:

- The function called by Simulink in a MATLAB Function block or by Stateflow in a MATLAB function.
- The function that you provide on the command line to codegen

For example, the following code generates compilation errors:

```
#!/codegen
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

To fix the problem, write a top-level function that specifies a fixed number of inputs and outputs and then call `inch_2_cm` as an external function or local function, as in this example:

```
#!/codegen
function [cmL, cmW, cmH] = conv_2_metric(inL, inW, inH)
[cmL, cmW, cmH] = inch_2_cm(inL, inW, inH);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

Use curly braces {} to index into the argument list

For code generation, you can use curly braces {}, but not parentheses (), to index into varargin and varargout arrays. For more information, see “Supported Index Expressions” on page 12-3.

Verify that indices can be computed at compile time

If you use an expression to index into `varargin` or `varargout`, make sure that the value of the expression can be computed at compile time. For examples, see “Apply Operations to a Variable Number of Arguments” on page 12-4.

Do not write to `varargin`

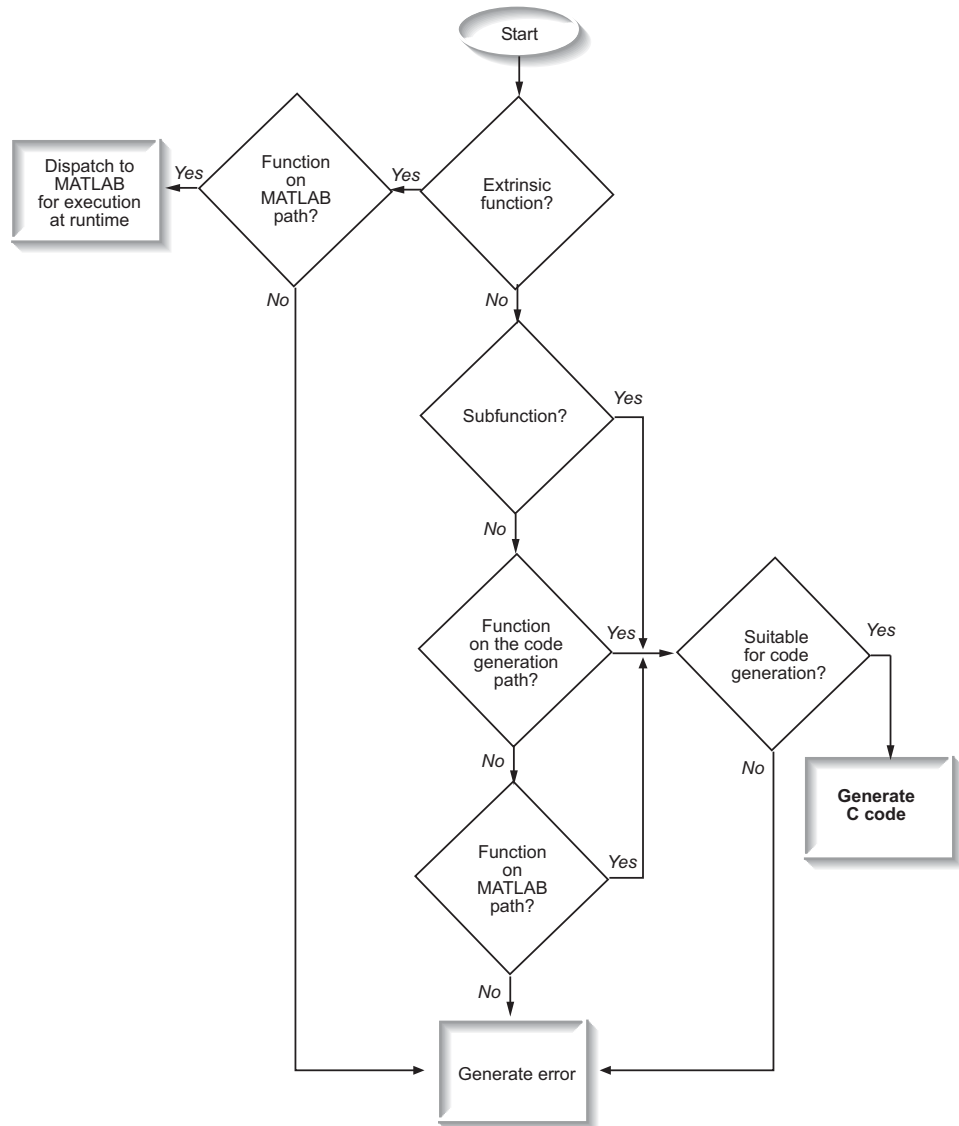
Generated code treats `varargin` as a read-only variable. If you want to write to input arguments, copy the values into a local variable.

Calling Functions for Code Generation

- “Resolution of Function Calls in MATLAB Generated Code” on page 13-2
- “Resolution of File Types on Code Generation Path” on page 13-6
- “Compilation Directive `%#codegen`” on page 13-8
- “Call Local Functions” on page 13-9
- “Call Supported Toolbox Functions” on page 13-10
- “Call MATLAB Functions” on page 13-11

Resolution of Function Calls in MATLAB Generated Code

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:



Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path
See “Compile Path Search Order” on page 13-4.
- Attempts to compile functions unless the code generation software determines that it should not compile them or you explicitly declare them to be extrinsic.

If a MATLAB function is not supported for code generation, you can declare it to be extrinsic by using the construct `coder.extrinsic`, as described in “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12. During simulation, the code generation software generates code for the call to an extrinsic function, but does not generate the function’s internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using the `coder.extrinsic` function.

- Resolves file type based on precedence rules described in “Resolution of File Types on Code Generation Path” on page 13-6

Compile Path Search Order

During code generation, function calls are resolved on two paths:

1 Code generation path

MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.

2 MATLAB path

If the function is not on the code generation path, MATLAB searches this path.

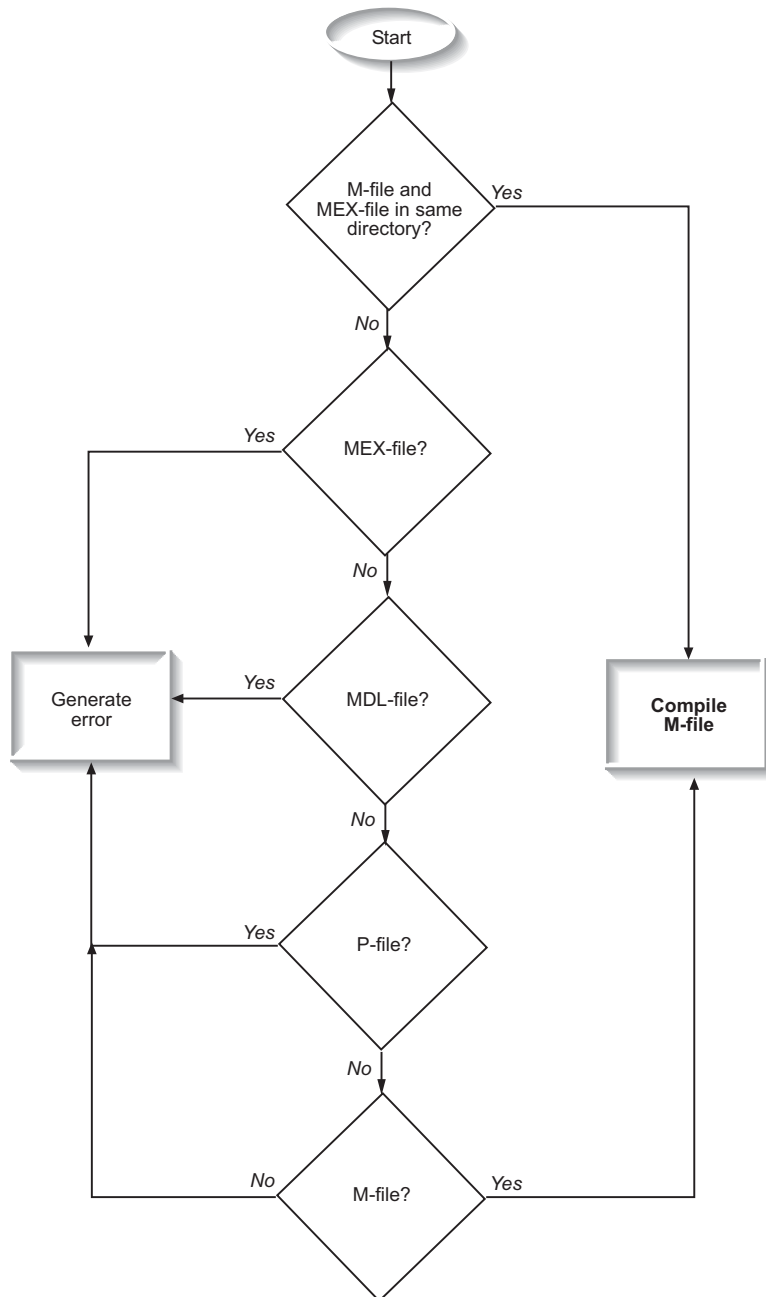
MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order”).

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

Resolution of File Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:



Compilation Directive `%#codegen`

Add the `%#codegen` directive (or pragma) to your function to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB code analyzer to help you diagnose and fix violations that would result in errors during code generation.

Call Local Functions

Local functions are functions defined in the body of MATLAB function. They work the same way for code generation as they do when executing your algorithm in the MATLAB environment.

The following example illustrates how to define and call a local function mean:

```
function [mean, stdev] = stats(vals)
%#codegen

% Calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals, '-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

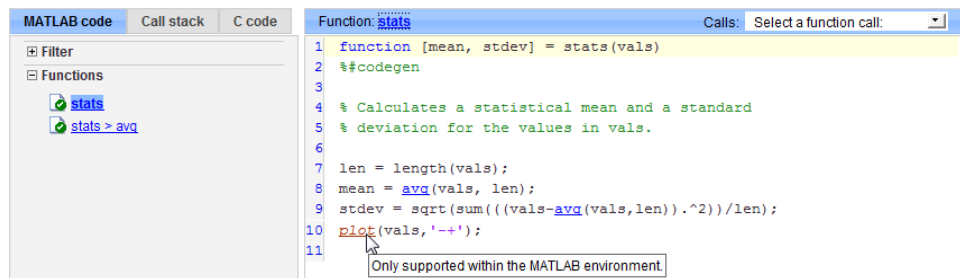
Call Supported Toolbox Functions

You can call toolbox functions directly if they are supported for code generation. For a list of supported functions, see “Functions Supported for Code Generation — Alphabetical List” on page 4-2.

Call MATLAB Functions

The code generation software attempts to generate code for functions, even if they are not supported for C code generation. The software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using `coder.extrinsic`. During simulation, the code generation software generates code for these functions, but does not generate their internal code. During standalone code generation, MATLAB attempts to determine whether the visualization function affects the output of the function in which it is called. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the visualization function from the generated code. Otherwise, compilation errors occur.

For example, you might want to call `plot` to visualize your results in the MATLAB environment. If you generate a MEX function from a function that calls `plot` and then run the generated MEX function, the code generation software dispatches calls to the `plot` function to MATLAB. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.



The screenshot shows the MATLAB IDE interface. On the left, the 'MATLAB code' pane displays a list of functions: 'stats' and 'stats > avg'. The main editor window shows the source code for the 'stats' function. The code is as follows:

```

1 function [mean, stdev] = stats(vals)
2 %#codegen
3
4 % Calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 len = length(vals);
8 mean = avg(vals, len);
9 stdev = sqrt(sum((vals-avg(vals,len)).^2)/len);
10 plot(vals, '-+');
11

```

The `plot` function call on line 10 is highlighted in yellow, and a tooltip points to it with the text "Only supported within the MATLAB environment".

For unsupported functions other than common visualization functions, you must declare the functions (like `pause`) to be extrinsic (see “Resolution of Function Calls in MATLAB Generated Code” on page 13-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 13-16).

There are two ways to declare a function to be extrinsic:

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 13-16).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

```
coder.extrinsic('function_name_1', ... , 'function_name_n');
```

Declaring Extrinsic Functions

The following code declares the MATLAB `patch` function extrinsic in the local function `create_plot`:

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
% and displays the triangle.
```

```
c = sqrt(a^2 + b^2);
create_plot(a, b, color);
```

```
function create_plot(a, b, color)
%Declare patch and axis as extrinsic
```

```
coder.extrinsic('patch');
```

```
x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```

The code generation software detects that `axis` is not supported for code generation and automatically treats it as an extrinsic function. The compiler does not generate code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

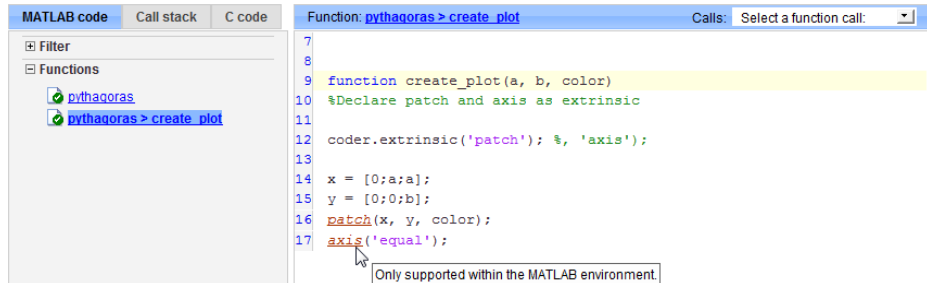
To test the function, follow these steps:

- 1 Convert `pythagoras` to a MEX function by executing this command at the MATLAB prompt:

```
codegen -report pythagoras -args {1, 1, [.3 .3 .3]}
```

- 2 Click the link to the code generation report and then, in the report, view the MATLAB code for `create_plot`.

The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.



The screenshot shows the MATLAB code generation report for the `pythagoras` function. The left pane shows the function list with `pythagoras > create_plot` selected. The right pane displays the MATLAB code for `create_plot`, with lines 16 and 17 highlighted in red. A tooltip points to these lines, stating "Only supported within the MATLAB environment."

```

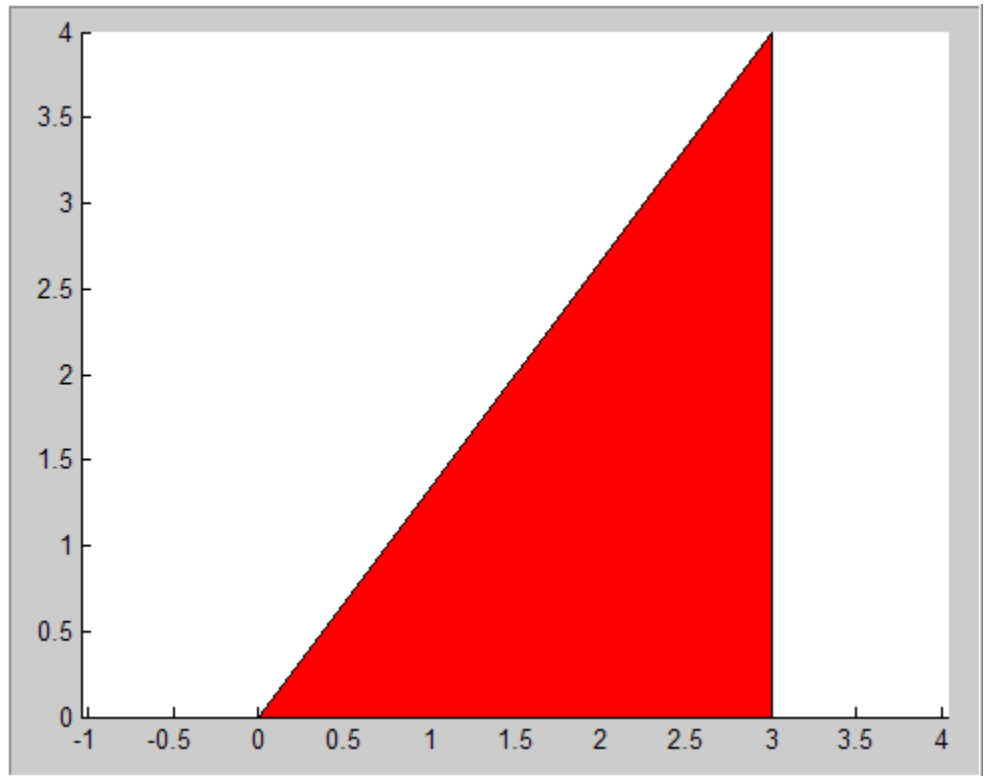
7
8
9 function create_plot(a, b, color)
10 %Declare patch and axis as extrinsic
11
12 coder.extrinsic('patch'); % 'axis');
13
14 x = [0;a;a];
15 y = [0;0;b];
16 patch(x, y, color);
17 axis('equal');

```

- 3 Run the MEX function by executing this command:

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays a plot of the right triangle as a red patch object:



When to Use the `coder.extrinsic` Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output — such as `pause` — during simulation, without generating unnecessary code (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 13-16).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with `mxArrays`” on page 13-17).
- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see “Scope of Extrinsic Function Declarations” on page 13-15).

- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 13-15). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 13-16).

Rules for Extrinsic Function Declarations

Observe the following rules when declaring functions extrinsic for code generation:

- Declare the function extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.

Scope of Extrinsic Function Declarations

The `coder.extrinsic` construct has function scope. For example, consider the following code:

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, `rat` and `min` are treated as extrinsic every time they are called in the main function `foo`. There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a local function, as in this example:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = mymin(N, D);

function y = mymin(a,b)
coder.extrinsic('min');
y = min(a,b);
```

Here, the function `rat` is extrinsic every time it is called inside the main function `foo`, but the function `min` is extrinsic only when called inside the local function `mymin`.

- Call the MATLAB function using `feval`, as described in “Calling MATLAB Functions Using `feval`” on page 13-16.

Calling MATLAB Functions Using `feval`

The function `feval` is automatically interpreted as an extrinsic function during code generation. Therefore, you can use `feval` to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

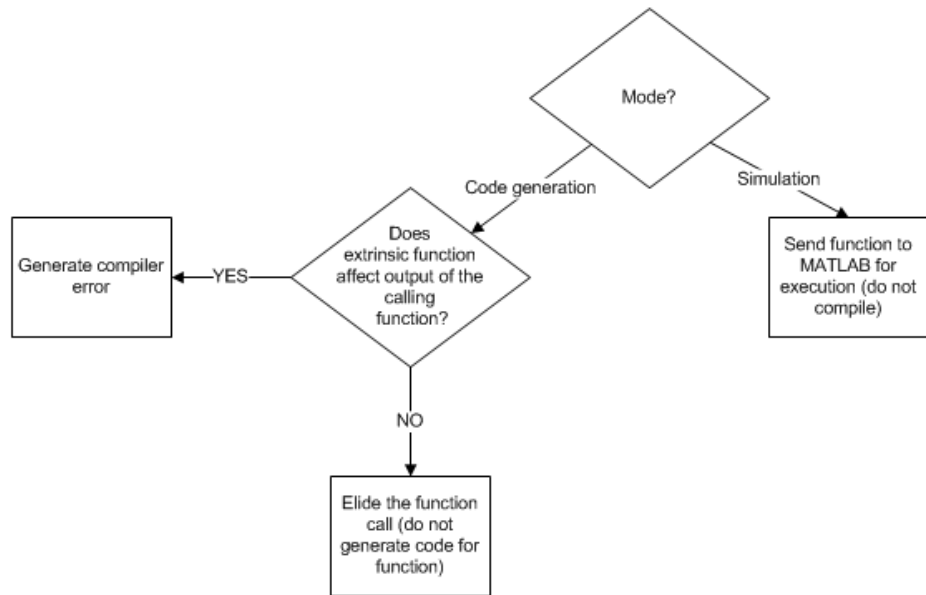
Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because `feval` is extrinsic, the statement `feval('min', N, D)` is evaluated by MATLAB — not compiled — which has the same result as declaring the function `min` extrinsic for just this one call. By contrast, the function `rat` is extrinsic throughout the function `foo`.

How MATLAB Resolves Extrinsic Functions During Simulation

MATLAB resolves calls to extrinsic functions — functions that do not support code generation — as follows:



During simulation, MATLAB generates code for the call to an extrinsic function, but does not generate the function’s internal code. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see “Working with `mxArrays`” on page 13-17). Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, MATLAB issues a compiler error.

Working with `mxArrays`

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables
- Passing `mxArrays` to functions and returning them from functions
- Converting `mxArrays` to known types at run time

To use `mxAArrays` returned by extrinsic functions in other operations, you must first convert them to known types, as described in “Converting `mxAArrays` to Known Types” on page 13-18.

Converting `mxAArrays` to Known Types

To convert an `mxAArray` to a known type, assign the `mxAArray` to a variable whose type is defined. At run time, the `mxAArray` is converted to the type of the variable assigned to it. However, if the data in the `mxAArray` is not consistent with the type of the variable, you get a run-time error.

For example, consider this code:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = min(N, D);
```

Here, the top-level function `foo` calls the extrinsic MATLAB function `rat`, which returns two `mxAArrays` representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these `mxAArrays` to another MATLAB function — in this case, `min` — you cannot assign the `mxAArray` returned by `min` to the output `y`.

If you run this function `foo` in a MATLAB Function block in a Simulink model, the code generates the following error during simulation:

Function output 'y' cannot be of MATLAB type.

To fix this problem, define `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```


Restrictions on Extrinsic Functions for Code Generation

The full MATLAB run-time environment is not supported during code generation. Therefore, the following restrictions apply when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller, or read or write to the caller's workspace do not work during code generation. Such functions include:
 - `dbstack`
 - `evalin`
 - `assignin`
 - `save`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code may produce unpredictable results if your extrinsic function performs the following actions at run time:
 - Change folders
 - Change the MATLAB path
 - Delete or add MATLAB files
 - Change warning states
 - Change MATLAB preferences
 - Change Simulink parameters

Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.

Fixed-Point Conversion

- “Convert MATLAB Code to Fixed-Point C Code” on page 14-2
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-3
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-20
- “Specify Type Proposal Options” on page 14-34
- “Log Histogram Data” on page 14-37
- “View and Modify Variable Information” on page 14-40
- “Build Instrumented MEX Function” on page 14-44
- “Propose Fixed-Point Data Types” on page 14-45
- “Apply Fixed-Point Data Types” on page 14-54
- “Modify Data Type Proposal Settings” on page 14-60
- “Modify Instrumentation Report Settings” on page 14-64
- “Automated Fixed-Point Conversion” on page 14-65
- “Instrumented MEX Functions” on page 14-73

Convert MATLAB Code to Fixed-Point C Code

- 1** Create a MATLAB Coder project, add the entry-point function from which you want to generate code, and then define entry-point input types.
- 2** On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**. Then on the project **Fixed-Point Conversion** pane, click **Define and validate fixed-point types** to open the Fixed-Point Conversion tool.
- 3** Compute ranges by either simulating using a test file, using static analysis to derive ranges from design ranges, or both.
- 4** Validate the proposed data types. See “Validating Types” on page 14-72.
- 5** Test numerics. See “Testing Numerics” on page 14-72.
- 6** In the MATLAB Coder project, select the **Build** tab, set the **Output type** to build a static or dynamic library, or executable, and then click **Build**.

MATLAB Coder generates fixed-point C code for your entry-point MATLAB function.

For more information, see “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-3 and “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-20.

Propose Fixed-Point Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Before generating C code, you must set up the C compiler. See “Setting Up the C/C++ Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```
- 3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	fun_with_matlab.m	Entry-point MATLAB function
Test file	fun_with_matlab_test.m	MATLAB script that tests fun_with_matlab.m

The fun_with_matlab Function

```
function y = fun_with_matlab(x) %#codegen
persistent z
if isempty(z)
    z = zeros(2,1);
end
% [b,a] = butter(2, 0.25)
b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
a = [
        1, -0.942809041582063, 0.333333333333333];

y = zeros(size(x));
for i=1:length(x)
    y(i) = b(1)*x(i) + z(1);
    z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
    z(2) = b(3)*x(i)          - a(3) * y(i);
end
end
```

The fun_with_matlab_test Script

The test script runs the fun_with_matlab function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```
% fun_with_matlab_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
```

```
x_step = ones(1,N);          % Step
x_impulse = zeros(1,N);     % Impulse
x_impulse(1)=1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i=1:size(x,1)
    y(i,:) = fun_with_matlab(x(i,:));
end

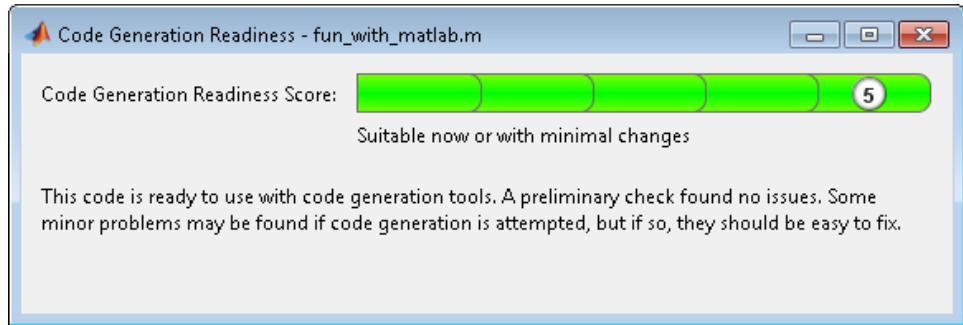
% Plot the results
titles = {'Chirp','Step','Impulse'};
clf
for i=1:size(x,1)
    subplot(size(x,1),1,i);
    plot(t,x(i,:),t,y(i,:));
    title(titles{i})
    legend('Input','Output');
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.');
```

Check Code Generation Readiness

In the current working folder, right-click the `fun_with_matlab.m` function. From the context menu, select **Check Code Generation Readiness**.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `fun_with_matlab.m` function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis”.

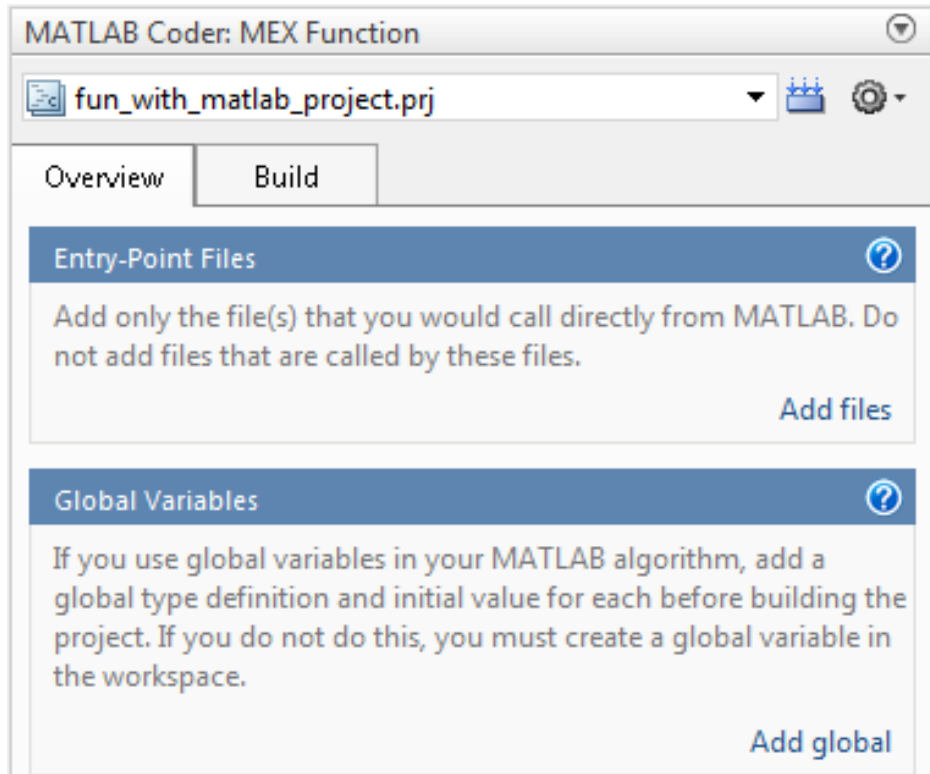
Create and set up a MATLAB Coder Project

- 1** Navigate to the work folder that contains the file for this example.
- 2** On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `fun_with_matlab_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_matlab_project.prj
```

By default, the project opens in the MATLAB workspace.

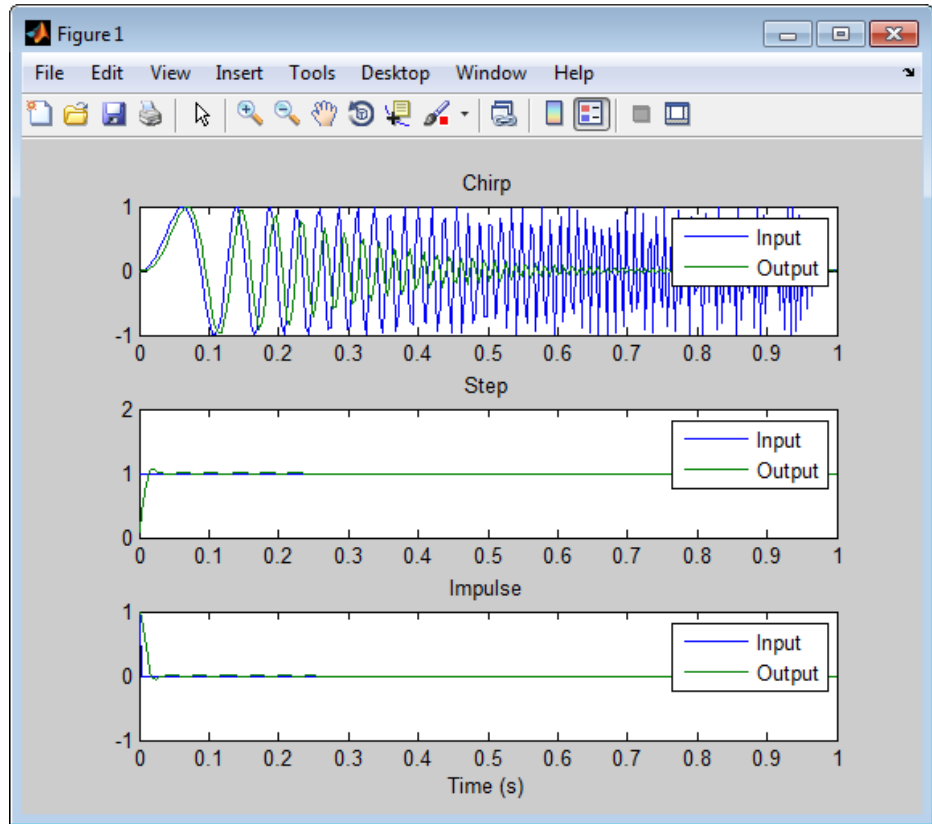


- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

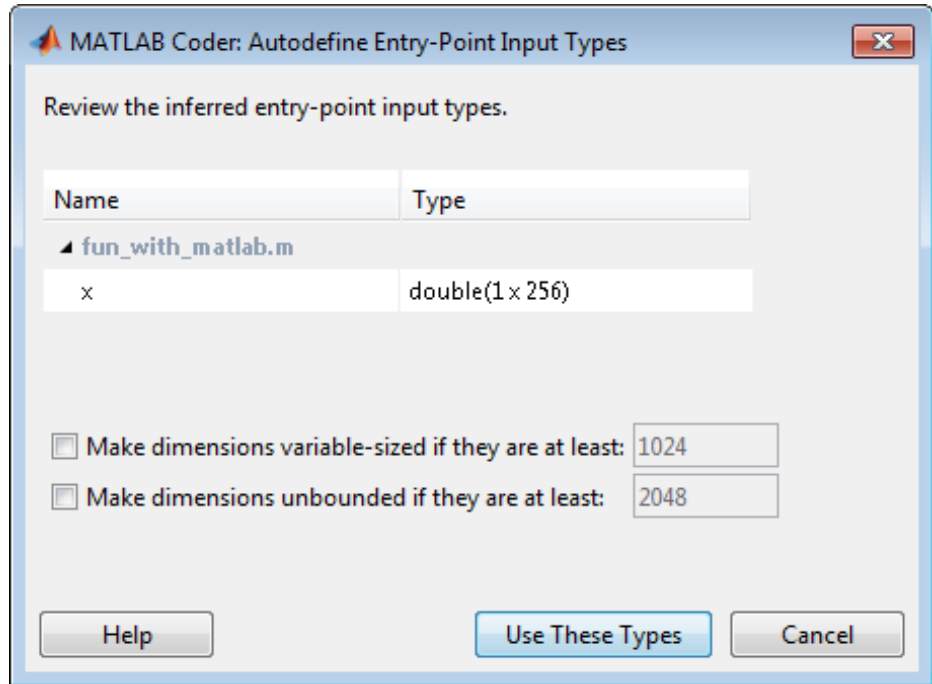
Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `fun_with_matlab_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



MATLAB Coder determines the input types from the test file and then displays them.

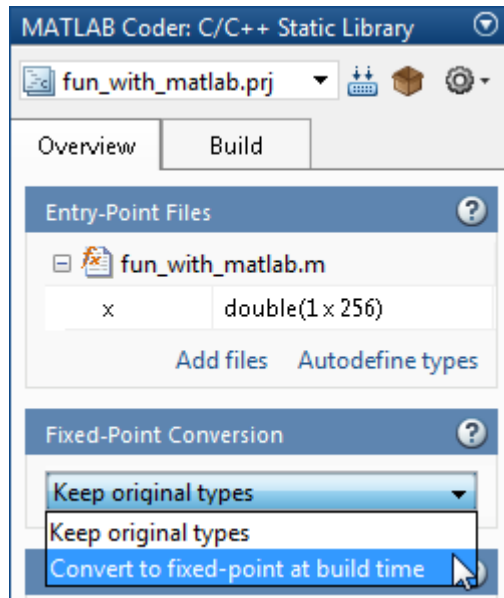


3 In the Autodefine Input Types dialog box, click **Use These Types**.

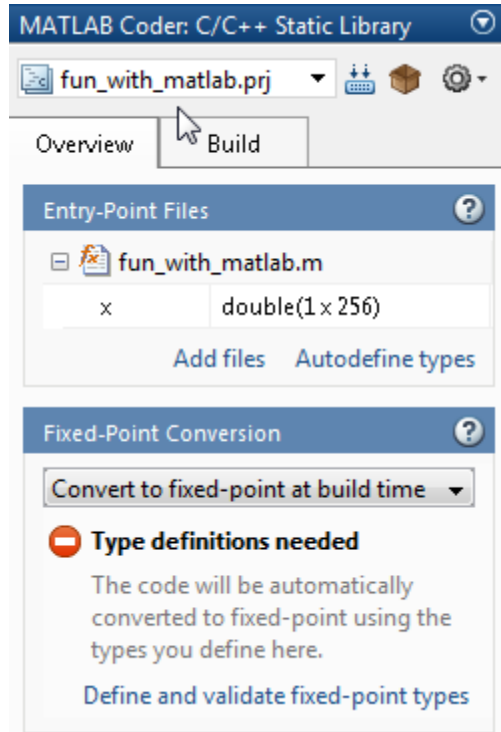
MATLAB Coder sets the type of `x` to `double(1x256)`.

Fixed-Point Conversion

1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

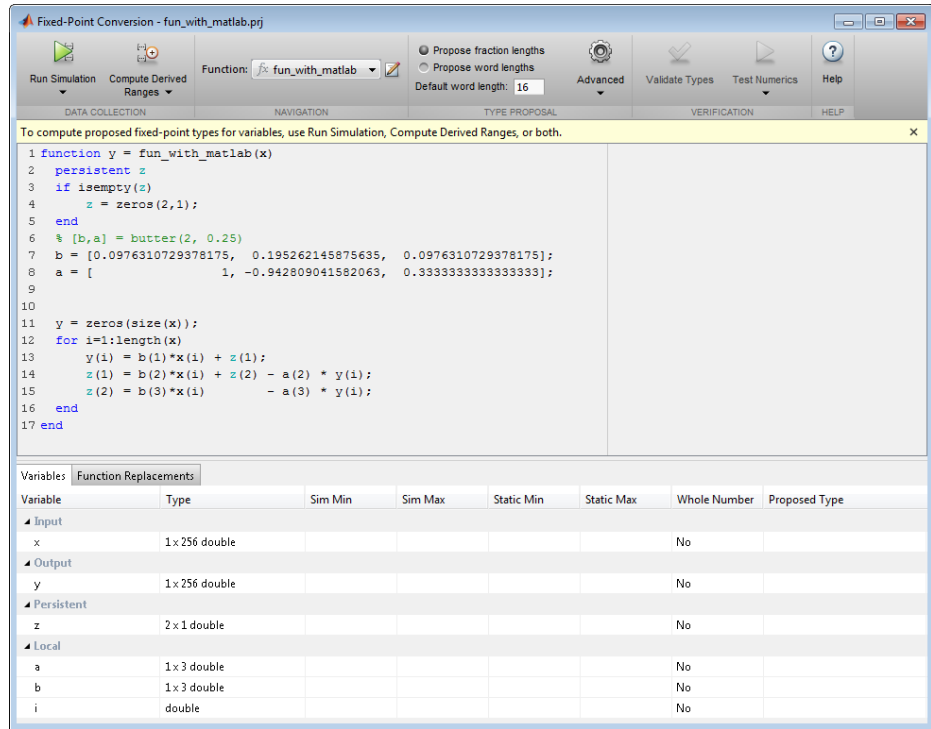


The project indicates that you must first define the fixed-point data types.



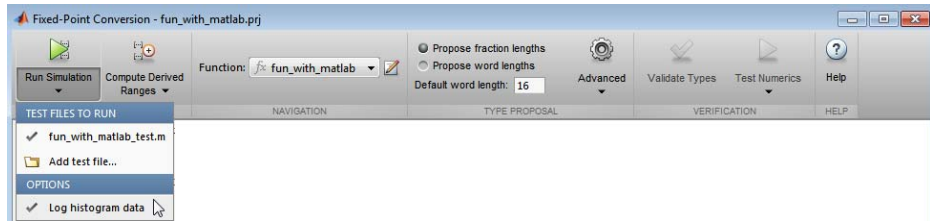
- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information” on page 14-40.

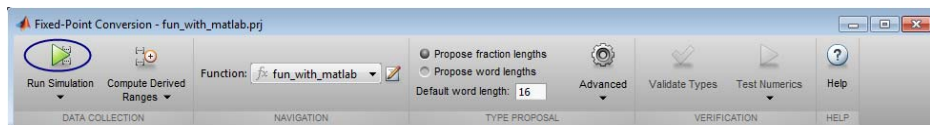


If the MEX function generation fails, the tool provide error message links to help you navigate to the code that caused the build issues. If your code contains functions that are not supported for fixed-point conversion, the tool displays these on the **Function Replacements** tab. For more information, see “Running a Simulation” on page 14-67.

- 3 Click **Run Simulation** and verify that the `fun_with_matlab_test` file is selected as a test file to run. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the conversion tool merges the simulation results. To clear results, right-click the **Variables** tab and select **Reset** entire table.
- 4 Click **Run Simulation** and select **Log histogram** data.



5 Click the Run Simulation button.



The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

The screenshot shows the 'Fixed-Point Conversion - fun_with_matlab.prj' window. The MATLAB code is as follows:

```

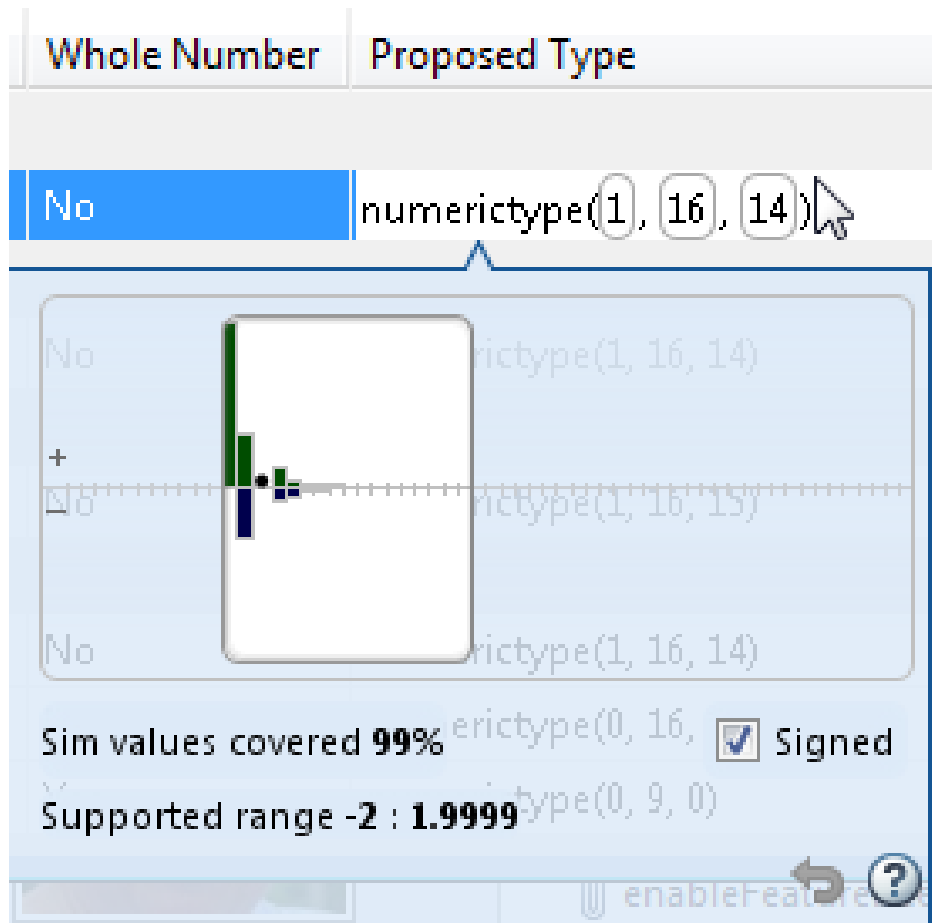
1 function y = fun_with_matlab(x)
2     persistent z
3     if isempty(z)
4         z = zeros(2,1);
5     end
6     % [b,a] = butter(2, 0.25)
7     b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
8     a = [1, -0.942809041582063, 0.333333333333333];
9
10
11    y = zeros(size(x));
12    for i=1:length(x)
13        y(i) = b(1)*x(i) + z(1);
14        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15        z(2) = b(3)*x(i) - a(3) * y(i);
16    end
17 end
    
```

Below the code is a table with the following data:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double	-0.9999756307053946	1			No	numerictype(1, 16, 14)
Output							
y	1 x 256 double	-0.97	1.06			No	numerictype(1, 16, 14)
Persistent							
z	2 x 1 double	-0.89	0.96			No	numerictype(1, 16, 15)
Local							
a	1 x 3 double	-0.94	1			No	numerictype(1, 16, 14)
b	1 x 3 double	0.1	0.2			No	numerictype(0, 16, 18)
i	double	1	256			Yes	numerictype(0, 9, 0)

If a value has . . . next to it, the value is rounded. Place your cursor over the . . . to view the actual value.

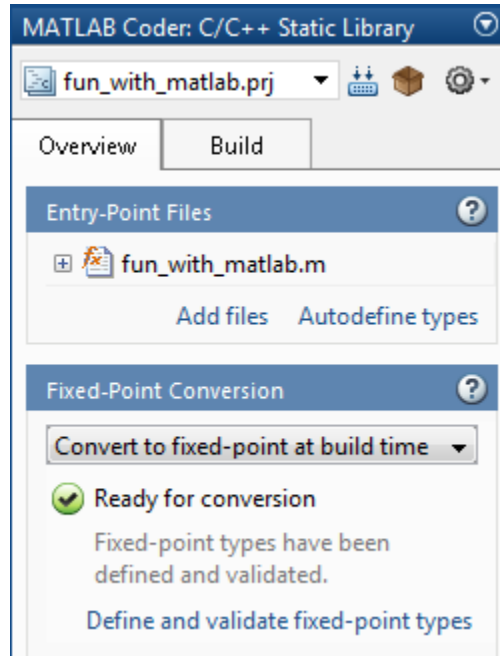
- Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.



To modify the proposed data types, either enter the required type into the **ProposedType** field or use the histogram controls. For more information about the histogram, see “Histogram” on page 14-70.

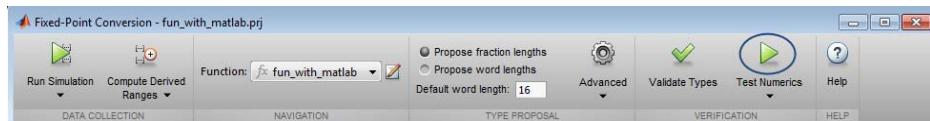
- 7 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a **Validation succeeded** message, and enables the **Test Numerics** option. The project indicates that you have validated the fixed-point data types.



If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see “Validating Types” on page 14-72.

- 8 Click **Test Numerics**, select Log inputs and outputs for comparison plots, and then click the Test Numerics button.



The tool runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable y . Because you selected to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output.

The maximum error is less than 0.03%. For the purpose of this example, this margin of error is acceptable, so you are ready to generate fixed-point C code.

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics” on page 14-72.

- 9 Return to the MATLAB Coder project.

Generate Fixed-Point C Code

- 1 In the MATLAB Coder project, verify that the **Fixed-Point Conversion** pane displays **Ready for conversion**, and then select the **Build** tab.
- 2 On this tab, set the **Output type** to **C/C++ Static library**.

The default output file name is `fun_with_matlab`.

- 3 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/fun_with_matlab_FixPt`.

- 4 To view the generated code, click **View report**.

The code generation report opens and displays the generated code for `fun_with_matlab_FixPt.c`. In the generated C code, the variables are not assigned real types, they are assigned fixed-point data types.

In this case, the generated code is not optimized; it contains a number of utility functions, such as `MultiWordAdd`. MATLAB Coder generates these utility functions because the results of adding or multiplying inputs results in a sum or product that exceed 32 bits. You can optimize the generated code by modifying the word length and `fimath` settings.

Optimize Fixed-Point C Code

- 1 In the Fixed-Point Conversion tool, click **Advanced** to display the advanced type proposal settings.

The fimath **Product mode** and **Sum mode** settings are both set to FullPrecision. In FullPrecision mode, the product word length grows to the sum of the word lengths of the operands. This causes the pro

2 Set the fimath **Product mode** and **Sum mode** to SpecifyPrecision.

Selecting SpecifyPrecision enables the Product word length, Product fraction length, Sum word length, and Sum fraction length settings. The product word length and sum word length are both set to 32, which limits these word lengths to 32 in the generated code.

3 Click **Validate Types**.

Because you have changed type proposal settings, you must validate the types again.

The software validates the proposed types, displays a Validation succeeded message.

4 Click the Test Numerics button.

The maximum error is still less than 0.03%, so you are ready to generate fixed-point C code.

5 Generate code again and view the generated C code for fun_with_matlab_FixPt.c. This time, because the word lengths in the generated code do not exceed 32 bits, the generated code does not contain utility functions.

```
void fun_with_matlab_FixPt(const int16_T x[256], int16_T y[256])
{
    int32_T i0;
    int32_T i;
    int32_T i1;
    int16_T b_y;
    int32_T i2;
    int32_T i3;
    /* [b,a] = butter(2, 0.25) */
    for (i0 = 0; i0 < 256; i0++) {
        y[i0] = 0;
    }
}
```

```
for (i = 0; i < 256; i++) {
    i0 = 25593 * x[i];
    if (i0 >= 0) {
        i1 = (int32_T)((uint32_T)i0 >> 4);
    } else {
        i1 = ~(int32_T)((uint32_T)~i0 >> 4);
    }

    i0 = i1 + (z[0] << 13);
    if (i0 >= 0) {
        b_y = (int16_T)((uint32_T)i0 >> 14);
    } else {
        b_y = (int16_T)~(int32_T)((uint32_T)~i0 >> 14);
    }

    i0 = 51186 * x[i];
    if (i0 >= 0) {
        i2 = (int32_T)((uint32_T)i0 >> 4);
    } else {
        i2 = ~(int32_T)((uint32_T)~i0 >> 4);
    }

    i0 = (i2 + (z[1] << 13)) - -15447 * b_y;
    if (i0 >= 0) {
        z[0] = (int16_T)((uint32_T)i0 >> 13);
    } else {
        z[0] = (int16_T)~(int32_T)((uint32_T)~i0 >> 13);
    }
}
```

Propose Fixed-Point Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges that you specify. The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time so you can save time by deriving ranges instead.

Prerequisites

Proposing data types based on derived ranges is not available on Mac platforms.

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Before generating C code, you must set up the C compiler. See “Setting Up the C/C++ Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver` .

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\dti`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

3 Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>dti.m</code>	Entry-point MATLAB function
Test file	<code>dti_test.m</code>	MATLAB script that tests <code>dti.m</code>

The `dti` Function

The `dti` function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
```

```
        clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;

function b = subFunction(a)
b = a*a;
```

The dti_test Function

The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10);

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
```



```

lower_limit = -500;

% call to the design that does DTI
[y_out(ii), is_clipped_out(ii)] = dti(data);

end

figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(1:len,x_in);
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

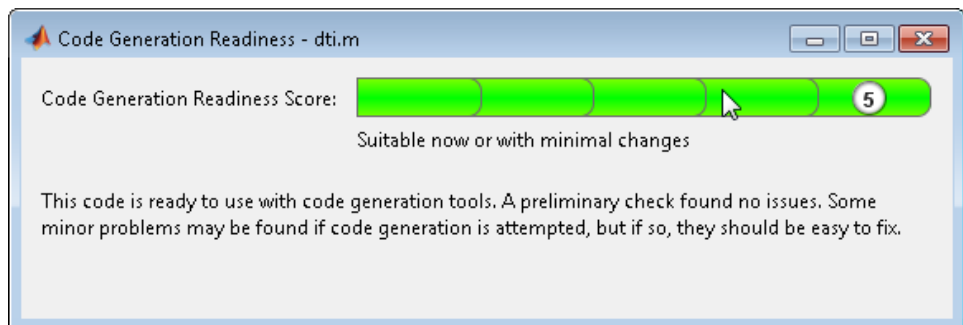
subplot(2,1,2); plot(1:len,y_out);
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (DTI)')

disp('Test complete.');
```

Check Code Generation Readiness

In the current working folder, right-click the `dti.m` function. From the context menu, select Check Code Generation Readiness.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `dti.m` function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis”.

Create and set up a MATLAB Coder Project

- 1** Navigate to the work folder that contains the file for this example.
- 2** On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `dti.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new dti.prj
```

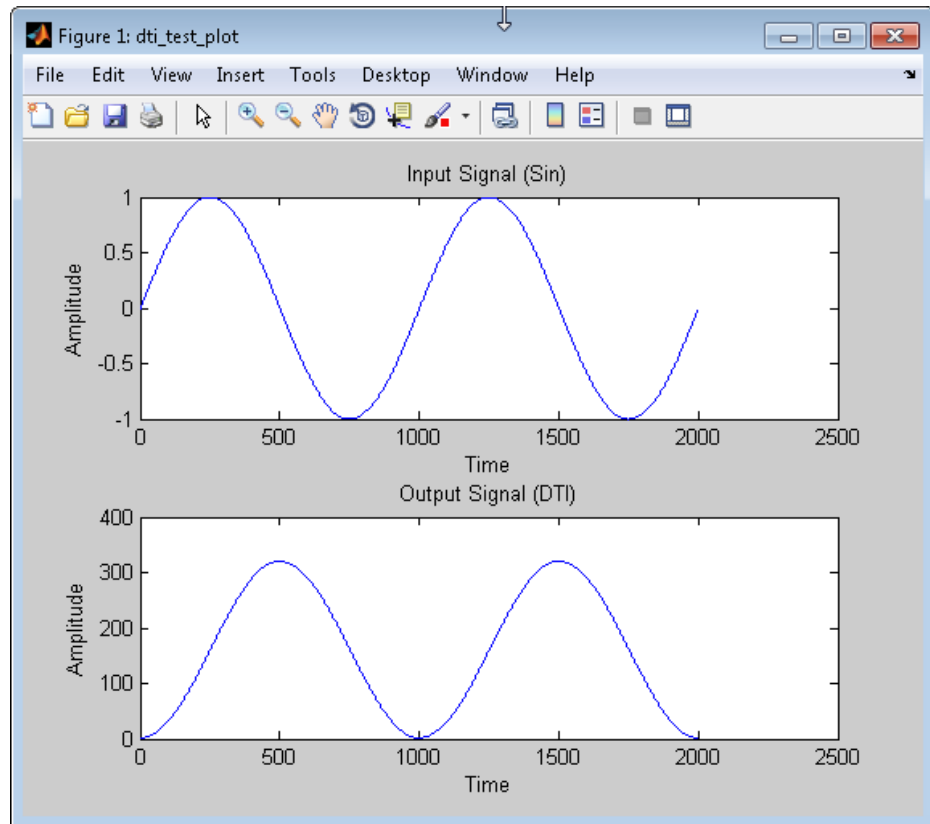
By default, the project opens in the MATLAB workspace.

- 3** On the project **Overview** tab, click the **Add files** link. Browse to the file `dti.m` and then click **OK** to add the file to the project.

Define Input Types

- 1** On the project **Overview** tab, click the **Autodefine types** link.
- 2** In the Autodefine Input Types dialog box, add `dti_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



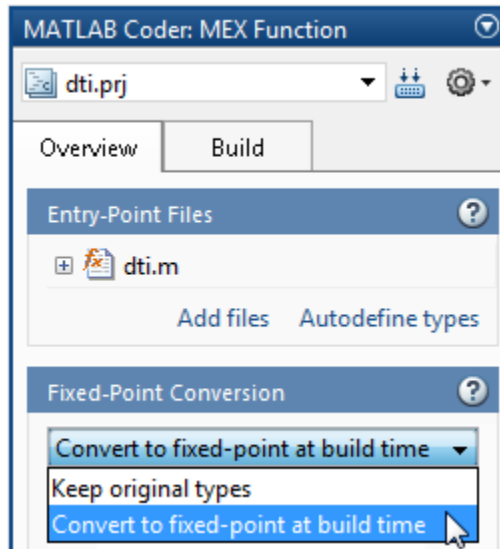
MATLAB Coder determines the input types from the test file and then displays them.

- 3** In the Autodefine Input Types dialog box, click **Use These Types**.

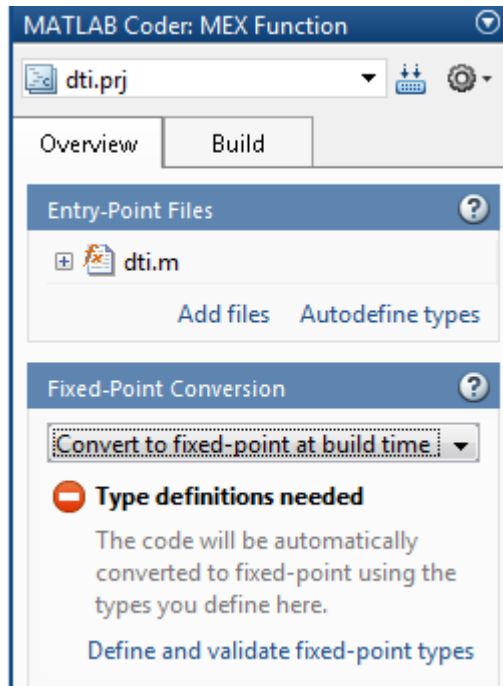
MATLAB Coder sets the type of `x` to `double(1x1)`.

Fixed-Point Conversion

- 1** On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

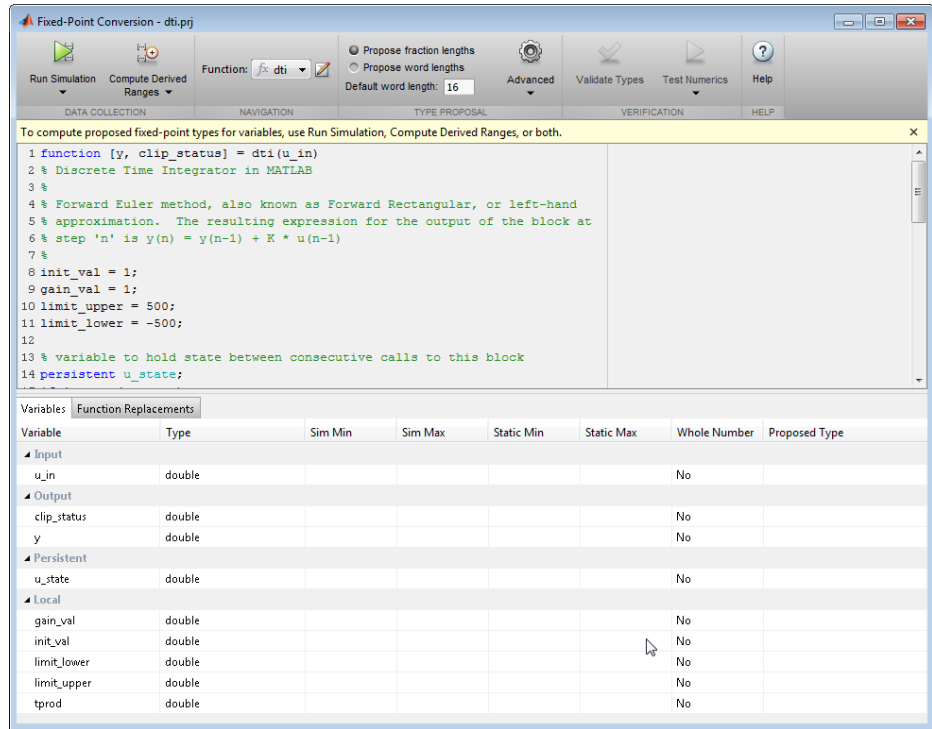


The project indicates that you must first define the fixed-point data types.



- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information” on page 14-40.



If the MEX function generation fails, the tool provide error message links to help you navigate to the code that caused the build issues. If your code contains functions that are not supported for fixed-point conversion, the tool displays these on the **Function Replacements** tab. For more information, see “Running a Simulation” on page 14-67.

- 3 In the Fixed-Point Conversion window, on the **Variables** tab, for input `u_in`, select **Static Min** and set it to -1. Then set **Static Max** to 1.

To compute derived range information, at a minimum you must specify static minimum and maximum values for all input variables. Alternatively, if you know what data type your hardware target uses, set the proposed data type to match this type.

4 Click the Compute Derived Ranges button.

Range analysis computes the derived ranges and displays them in the **Variables** tab. Using these derived ranges, the analysis proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

In the `dti` function, the `clip_status` output has a minimum value of -2 and a maximum of 2.

```
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```

When you derive ranges, the Fixed-Point Conversion tool analyses the function and computes these minimum and maximum values for `clip_status`.

The screenshot shows the 'Fixed-Point Conversion - dti.pj' window. The top toolbar includes 'Run Simulation', 'Compute Derived Ranges', 'Function: dti', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar are tabs for 'DATA COLLECTION', 'NAVIGATION', 'TYPE PROPOSAL', 'VERIFICATION', and 'HELP'. The main area contains MATLAB code for a discrete-time integrator. Below the code is a table titled 'Variables' with sub-tabs for 'Function Replacements' and 'Static Analysis: Output'.

```

1 function [y, clip_status] = dti(u_in) %#codegen
2 % Discrete Time Integrator in MATLAB
3 %
4 % Forward Euler method, also known as Forward Rectangular, or left-hand
5 % approximation. The resulting expression for the output of the block at
6 % step 'n' is y(n) = y(n-1) + K * u(n-1)
7 %
8 init_val = 1;
9 gain_val = 1;
10 limit_upper = 500;
11 limit_lower = -500;
12
13 % variable to hold state between consecutive calls to this block
14 persistent u_state;
15 if isempty(u_state)
16     u_state = init_val+1;

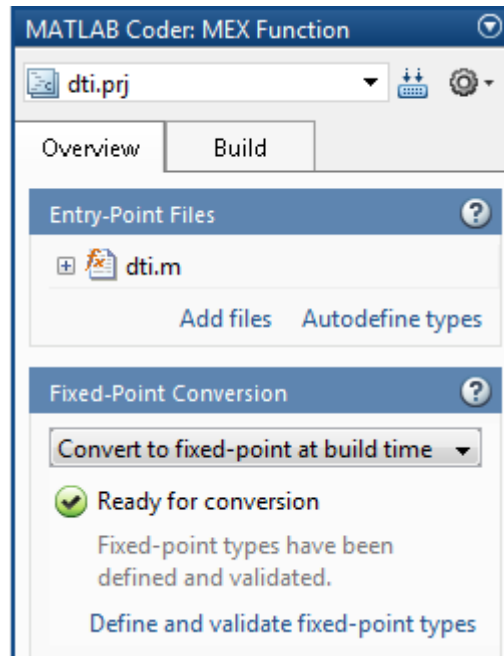
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u_in	double			-1	1	No	numerictype(1, 16, 14)
Output							
clip_status	double			-2	2	No	numerictype(1, 16, 13)
y	double			-500	500	No	numerictype(1, 16, 6)
Persistent							
u_state	double			-501	501	No	numerictype(1, 16, 6)
Local							
gain_val	double			1	1	Yes	numerictype(0, 1, 0)
init_val	double			1	1	Yes	numerictype(0, 1, 0)
limit_lower	double			-500	-500	Yes	numerictype(1, 10, 0)
limit_upper	double			500	500	Yes	numerictype(0, 9, 0)
tprod	double			-1	1	No	numerictype(1, 16, 14)

The tool provides a **Quick derived range analysis** option to select and the option to specify a timeout in case the analysis takes a very long time. For more information, see “Computing Derived Ranges” on page 14-68

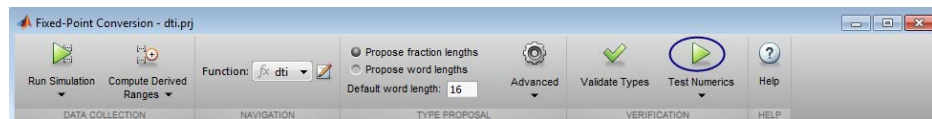
5 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a **Validation succeeded** message, and enables the **Test Numerics** option. The project indicates that you have validated the fixed-point data types.



If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see “Validating Types” on page 14-72.

- 6 Run the test file to test the fixed-point MATLAB code. Click **Test Numerics** and select Log inputs and outputs for comparison plots, and then click the Test Numerics button.

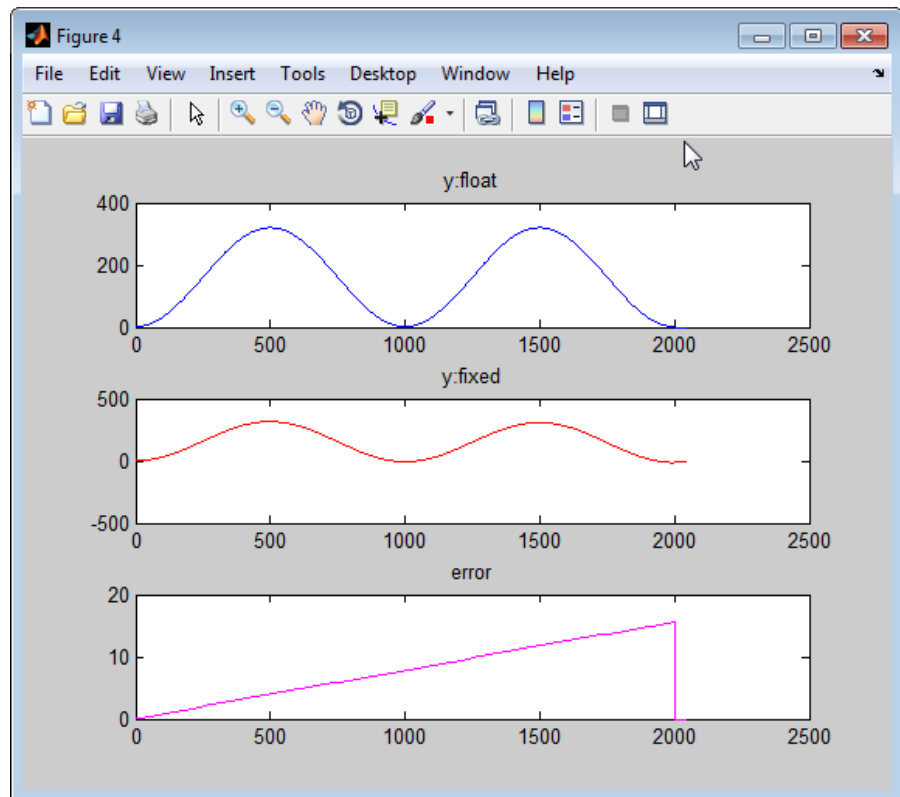


The tool runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variables `y` and `clip_status`. Because you selected to log

inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output.

Plots are displayed for the:

- Floating-point input and output signals.
- Fixed-point input and output signals.
- Outputs `y` and `clip_status` showing the difference between the floating-point and the fixed-point runs.



The maximum difference between the floating-point and fixed-point runs for `y` is less than 5%. For the purpose of this example, this margin of error is acceptable, so you are ready to generate fixed-point C code.

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics” on page 14-72.

Generate Fixed-Point C Code

1 In the MATLAB Coder project, select the **Build** tab.

2 On this tab, set the **Output type** to `C/C++ Static library`.

The default output file name is `dti`.

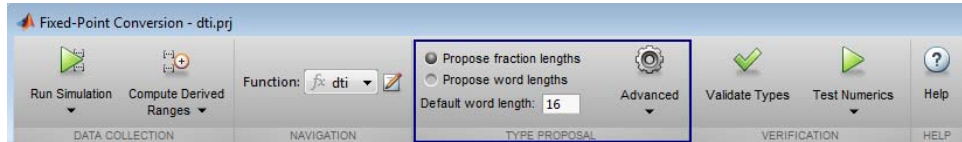
3 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/dti_FixPt`.

4 To view the generated code, click **View report**.

The code generation report opens and displays the generated code for `dti_FixPt.c`. In the generated C code, variables are assigned fixed-point data types.

Specify Type Proposal Options



You specify whether to propose fraction lengths or word lengths in the Fixed-Point Conversion window **Type Proposal** options. By default, the software proposes fraction lengths for a default word length of 16.

To customize fixed-point type proposals, use the **Advanced** settings.

Advanced Setting	Values	Description
When proposing types	ignore simulation ranges	Propose data types based only on derived ranges
	ignore derived ranges	Propose data types based only on simulation ranges
	use all collected data (default)	Proposed data types based on both simulation and derived ranges
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.

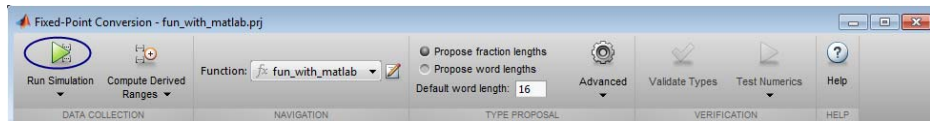
Advanced Setting	Values	Description
Safety margin for sim min/max (%)	0 (default)	<p>Specify safety factor for simulation minimum and maximum values.</p> <p>The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of 55 specifies that you want a range at least 55 percent larger. A value of -15 specifies that a range up to 15 percent smaller is acceptable.</p>
Generated fixed-point file name suffix	_FixPt (default)	<p>Specify the suffix to add to the generated fixed-point file names. For example, by default, if you generate a static library for a project named <code>test</code>, the generated files are in the subfolder <code>codegen\lib\test_FixPt</code>. The generated static library is named <code>test.lib</code>, but the generated C code files use the suffix, for example, <code>test_FixPt.c</code>.</p>
Transform for-loop index variables	No (default)	
	Yes	

Advanced Setting		Values	Description
fimath	Rounding method	Ceiling	Specify the fimath properties for the generated fixed-point data types. The default fixed-point math properties use the Floor rounding and Wrap overflow because they are the default actions in C. These settings generate the most efficient code but might cause problems with overflow.
		Convergent	
		Floor (default)	
		Nearest	
		Round	
		Zero	
	Overflow action	Saturate	After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification.
		Wrap (default)	
	Product mode	FullPrecision (default)	
		KeepLSB	
		KeepMSB	
		SpecifyPrecision	
	Sum mode	FullPrecision (default)	
		KeepLSB	
KeepMSB			
SpecifyPrecision			

Log Histogram Data

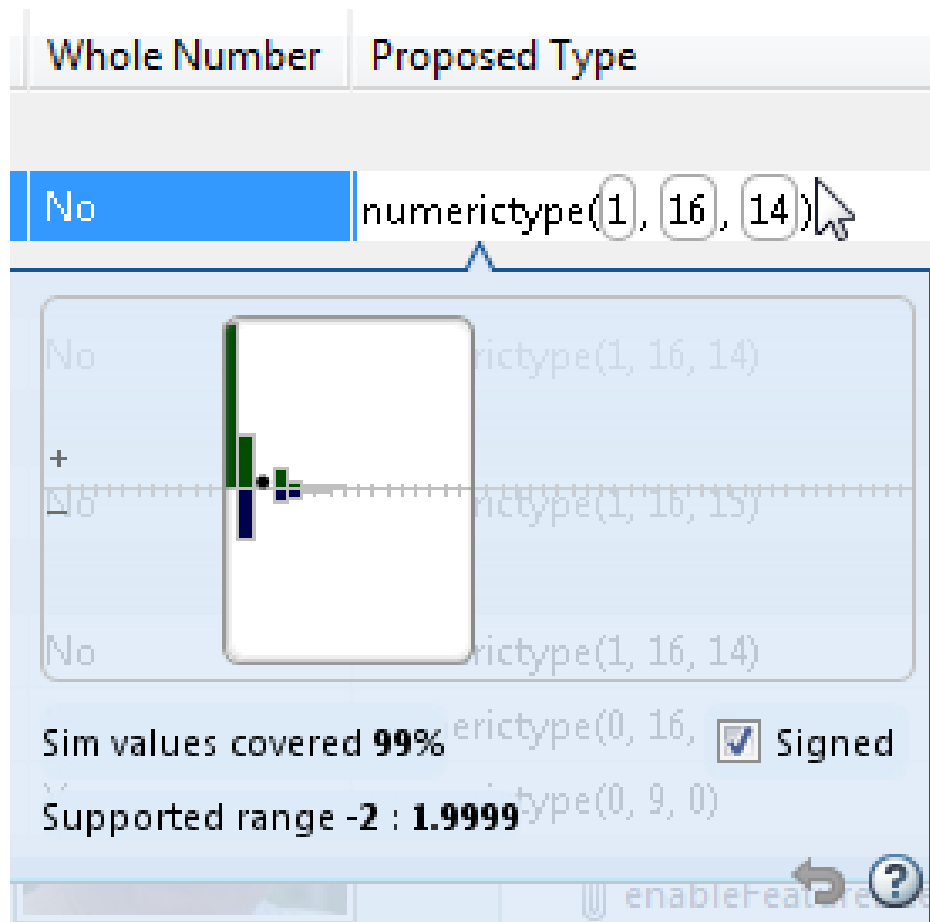
To log histogram data:

- 1 In the Fixed-Point Conversion window, click **Run Simulation** and select **Log histogram data**, and then click the Run Simulation button.

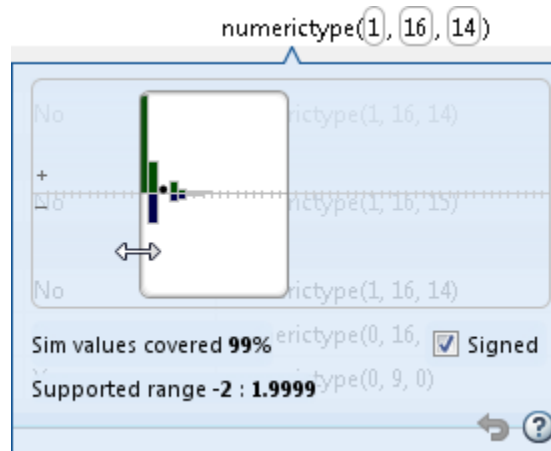


The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.


- 2 To view logged histogram data for a variable, click the variable's **Proposed Type** field.



- 3** You can view the effect of changing the proposed data types by:
- Selecting and dragging the white bounding box in the histogram window. This action does not change the word length of the proposed data type, but modifies the position of the binary point within the word so that the fraction length of the proposed data type changes.
 - Selecting and dragging the left edge of the bounding box to increase or decrease the word length. This action does not change the fraction length or the position of the binary point.



- Selecting and dragging the right edge to increase or decrease the fraction length of the proposed data type. This action does not change the position of the binary point. The word length changes to accommodate the fraction length.
- Selecting or clearing **Signed**. Clear **Signed** to ignore negative values.

Before committing changes, you can revert to the types proposed by the automatic conversion by clicking .

View and Modify Variable Information

View Variable Information

To view information about the variables in the MATLAB function selected in the **Navigation** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 14-69.

You can view the variable information:

- **Variable**
Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.
- **Type**
The original size, type, and complexity of each variable.
- **Sim Min**
The minimum value assigned to the variable during simulation.
- **Sim Max**
The maximum value assigned to the variable during simulation.

To search for variables in the MATLAB code pane and on the **Variables** tab, use **Ctrl+F**. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Modify Variable Information

If you modify variable information, the tool highlights the values in bold. You can modify the following fields:

- **Static Min**
You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 14-43.
Editing this field does not trigger static range analysis, but the tool uses the edited values in subsequent analyses.

- **Static Max**

You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 14-43.

Editing this field does not trigger static range analysis, but the tool uses the edited values in subsequent analyses.

- **Whole Number**

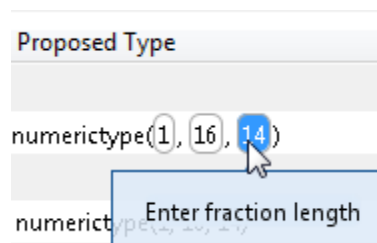
The Fixed-Point Conversion tool uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

Editing this field does not trigger static range analysis, but the tool uses the edited value in subsequent analyses.

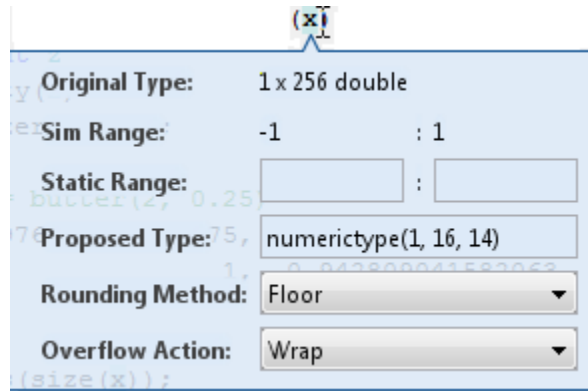
- **Proposed Type**

You can modify the signedness, word length, and fraction length settings individually by:

- On the **Variables** tab, by modifying the value in the **ProposedType** field.



- In the code window, by selecting a variable and then modifying the **ProposedType** field.



If you selected to log histogram data, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see “Histogram” on page 14-70.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select **Reset entire table**.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1x256 double	-1	1			No	numerictype(1, 16, 12)
Output							
y	1x256 double	-0.97	1.06				numerictype(1, 16, 14)
Persistent							
z	2x1 double	-0.89	0.96				numerictype(1, 16, 15)
Local							
a	1x3 double	-0.94	1				numerictype(1, 16, 14)
b	1x3 double	0.1	0.2			No	numerictype(0, 16, 18)
i	double	1	256			Yes	numerictype(0, 9, 0)

- To revert the type of a selected variable to the type computed by the tool, right-click the field and select **Undo changes**.
- To revert changes to variables, right-click the field and select **Undo changes for all variables**.
- To clear a static range value, right-click an edited field and select **Clear static range**.
- To clear manually-entered static range values, right-click anywhere on the **Variables** tab and select **Clear all manually entered static ranges**.

Promote Sim Min and Sim Max Values

The Fixed-Point Conversion tool allows you to promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.

Variables		Function Replacements					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Static Min	Static Max
▲ Input							
x	1 x 256 double	-1	1				
▲ Output							
y	1 x 256 double	-0.97	1.06				

Copy sim ranges for all top-level inputs
 Copy sim ranges for all persistent variables
 Clear all manually entered static ranges
 Reset entire table

To copy:

- A simulation range for a selected variable, select a variable, right-click and then select **Copy sim range**.
- Simulation ranges for top-level inputs, right-click the **Static Min** or **Static Max** column and then select **Copy sim ranges for all top-level inputs**.
- Simulation ranges for persistent variables, right-click the **Static Min** or **Static Max** column and then select **Copy sim ranges for all persistent inputs**.

Build Instrumented MEX Function

Note This capability is not compatible with automatic fixed-point conversion. If you select **Convert to fixed point at build time**, you cannot build instrumented MEX functions.

- 1** In the project, click the **Build** tab.
- 2** On the **Build** tab, set the **Output type** to Instrumented MEX Function.
- 3** Click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates an instrumented MEX function in the current folder. It also provides a link to the report on the **Show Instrumentation Results** pane. In this report, you can view the types of variables in your MATLAB code.

After you run the instrumented MEX function, the instrumentation report provides fixed-point data type proposals based on the simulation range data. You can use this information to convert your MATLAB code to fixed point by hand. For more information, see “Propose Fixed-Point Data Types” on page 14-45

Propose Fixed-Point Data Types

Note This capability is not compatible with automatic fixed-point conversion. If you select **Convert to fixed point** at build time, you cannot build instrumented MEX functions.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Before generating C code, you must set up the C compiler. See “Setting Up the C/C++ Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

- 3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	fun_with_matlab.m	Entry-point MATLAB function
Test file	fun_with_matlab_test.m	MATLAB script that tests fun_with_matlab.m

The fun_with_matlab Function

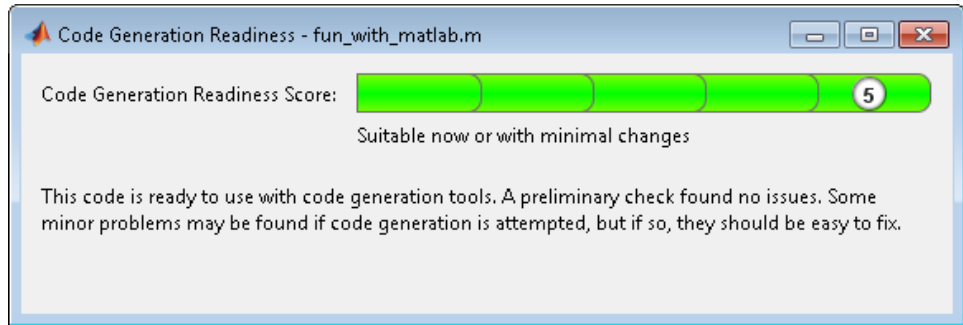
```
function y = fun_with_matlab(x) %#codegen
persistent z
if isempty(z)
    z = zeros(2,1);
end
% [b,a] = butter(2, 0.25)
b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
a = [
        1, -0.942809041582063, 0.333333333333333];

y = zeros(size(x));
for i=1:length(x)
    y(i) = b(1)*x(i) + z(1);
    z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
    z(2) = b(3)*x(i)          - a(3) * y(i);
end
end
```

Check Code Generation Readiness

In the current working folder, right-click the `fun_with_matlab.m` function. From the context menu, select **Check Code Generation Readiness**.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `fun_with_matlab.m` function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis”.

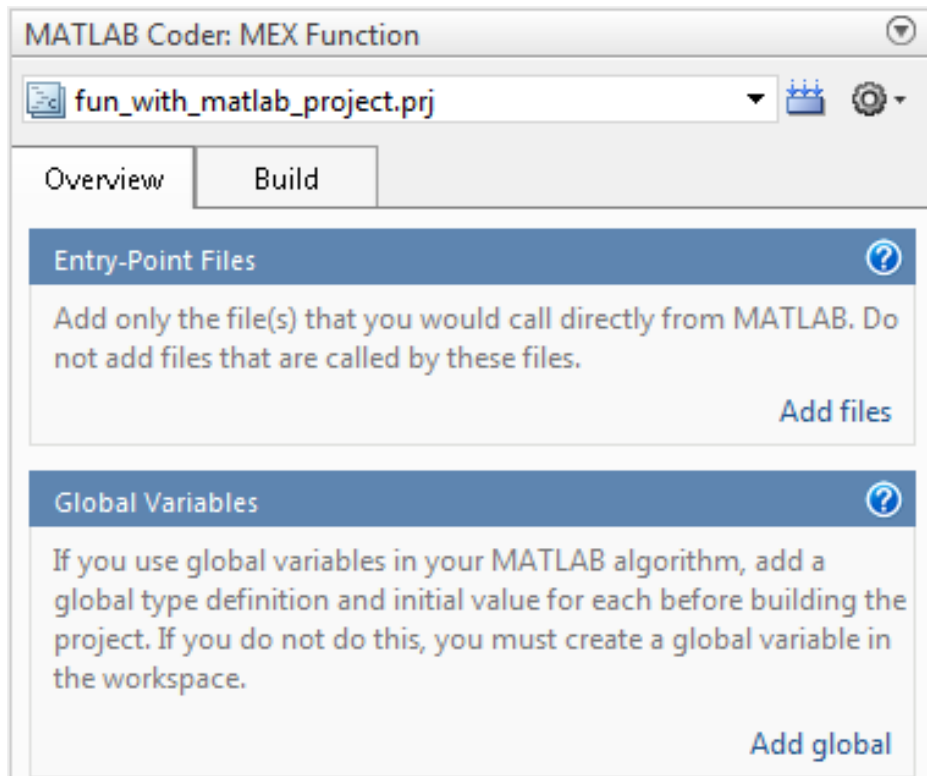
Create and set up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this tutorial.
- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_matlab_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_matlab_project.prj
```

By default, the project opens in the MATLAB workspace.



- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

About the `fun_with_matlab_test` Script

The test script runs the `fun_with_matlab` function with three input signals: `chirp`, `step`, and `impulse`. The script then plots the results.

Contents of `fun_with_matlab_test`

```
% fun_with_matlab_test
%
% Define representative inputs
N = 256;                % Number of points
t = linspace(0,1,N);   % Time vector from 0 to 1 second
```

```

f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1)=1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i=1:size(x,1)
    y(i,:) = fun_with_matlab(x(i,:));
end

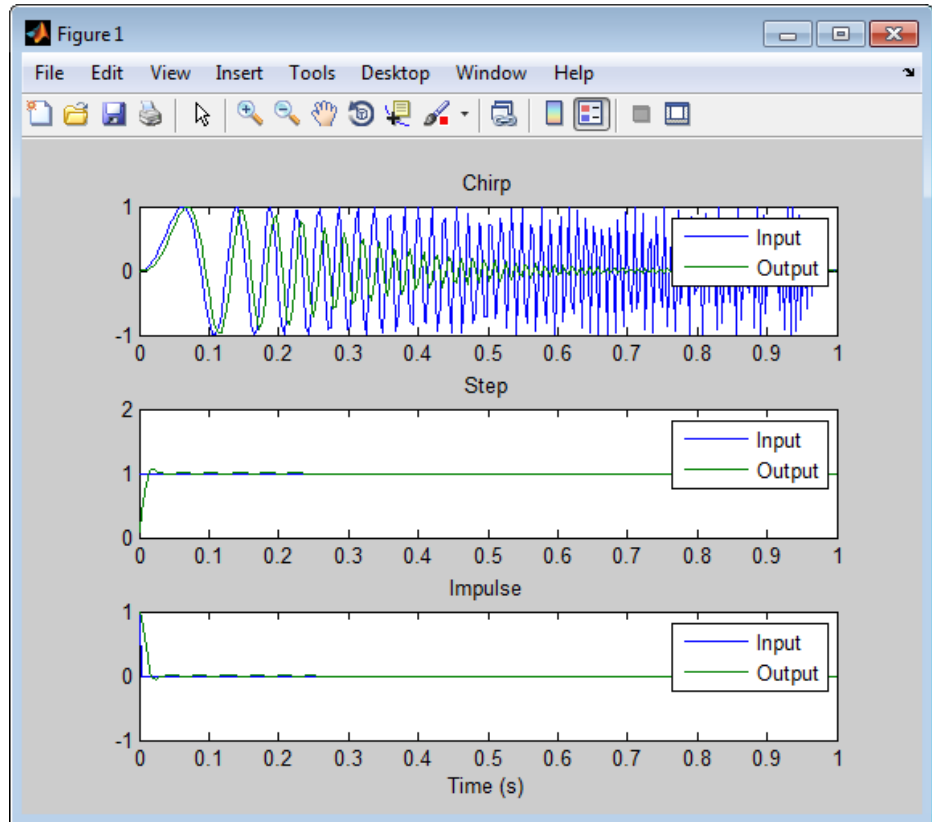
% Plot the results
titles = {'Chirp','Step','Impulse'};
clf
for i=1:size(x,1)
    subplot(size(x,1),1,i);
    plot(t,x(i,:),t,y(i,:));
    title(titles{i})
    legend('Input','Output');
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.');
```

Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `fun_with_matlab_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



MATLAB Coder determines the input types from the test file and then displays them in the Autodefine Input Types dialog box.

- 3 In this dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x256)`.

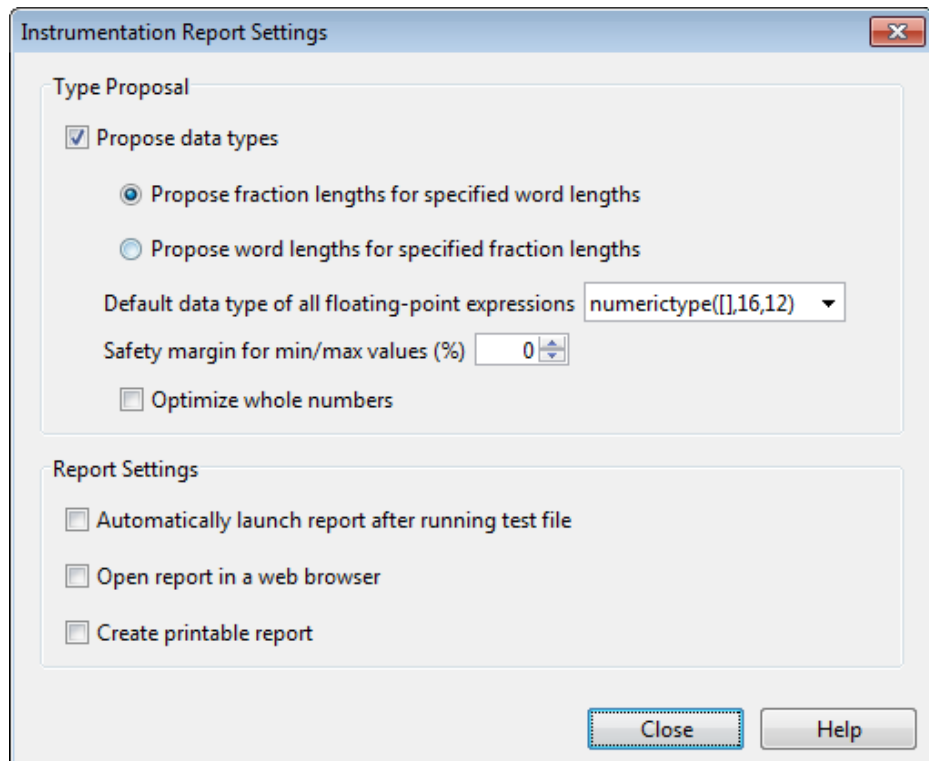
Build Instrumented MEX Function

- 1 In the project, click the **Build** tab.
- 2 On the **Build** tab, set the **Output type** to Instrumented MEX Function.
- 3 Click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates an instrumented MEX function `fun_with_matlab_mex` in the current folder. It also provides a link to the report on the **Show Instrumentation Results** pane. In this report, you can view the types of variables in your MATLAB code.

View Data Type Proposal Settings

- 1 On the **Show Instrumentation Results** pane, click the **Data type proposal and report settings** link.



This example uses the default data type proposal settings which propose fraction lengths for the specified word lengths. Because the MATLAB code is floating-point, the word length is specified by the **Default data type of all floating-point expressions** field. You can specify the `numerictype`

signedness, word length and fraction length. Specifying [] for signedness instructs MATLAB Coder to choose the signedness based on simulation values. The default word length is 16. The default fraction length is 12.

For more information, see “Modify Data Type Proposal Settings” on page 14-60.

- 2** Close the dialog box.

Run Simulation

- 1** On the **Run Simulation** pane, verify that the test file is set to `fun_with_matlab_test` and that **Redirect entry-point calls to MEX function** is selected. That way, each call to `fun_with_matlab` is replaced with a call to the instrumented MEX function `fun_with_matlab_mex`.
- 2** On the **Run Simulation** pane, click **Run**.

The `fun_with_matlab_test` file runs and calls `fun_with_matlab_mex`. The outputs of the filters are displayed as before.

View Code Generation Report

- 1** On the **Show Instrumentation Results** pane, click **View Report**.
- 2** In the **Code Generation Report**, click the **Variables** tab.

The report displays the simulation minimum and maximum values and the proposed data types.

The screenshot shows the MATLAB Code Generation Report for a function named 'fun_with_matlab'. The report includes the MATLAB code and a table of proposed fixed-point data types for the variables. The table is as follows:

Order	Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax
1	y	Output	1 x 256	double	No	Signed	16	14	No	-0.9696917930434206	1.0553496057969345
2	x	Input	1 x 256	double	No	Signed	16	14	No	-0.999756307053946	1
3	z	Persistent	2 x 1	double	No	Signed	16	15	No	-0.8907046852192462	0.957718532859117
4	h	Local	1 x 3	double	No	Unsigned	16	13	No	0.9976310729378175	0.195262145879635
5	a	Local	1 x 3	double	No	Signed	16	14	No	-0.942809041582083	1
6	l	Local	1 x 1	double	No	Unsigned	16	0	Yes	1	256

MATLAB Coder proposes data types with word length of 16 and fraction length optimized to avoid overflows.

Next Steps

To learn how to apply the proposed data types to your entry-point MATLAB function and verify that the fixed-point version of your algorithm is functionally equivalent to your original MATLAB algorithm, see “Apply Fixed-Point Data Types” on page 14-54.

Apply Fixed-Point Data Types

Note This capability is not compatible with automatic fixed-point conversion. If you select `Convert to fixed point` at build time, you cannot build instrumented MEX functions.

This example shows you how to write a fixed-point version of your entry-point function using the data types proposed in “Propose Fixed-Point Data Types” on page 14-45.

You will learn how to:

- Use the proposed data types to create a fixed-point version of your entry-point function.
- Update your test file to call the fixed-point entry-point function.
- Verify that the fixed-point function is functionally equivalent to the original MATLAB algorithm.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Before generating C code, you must set up the C compiler. See “Setting Up the C/C++ Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check

which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:


```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```
- 3 Copy the following files to your local working folder.

Type	Name	Description
Function code	<code>fun_with_matlab.m</code>	Entry-point MATLAB function
Test file	<code>fun_with_matlab_test.m</code>	MATLAB script that tests <code>fun_with_matlab.m</code>
Function code	<code>fun_with_fi.m</code>	Entry-point MATLAB function — fixed-point version of <code>fun_with_matlab</code> that uses data types proposed in “Propose Fixed-Point Data Types” on page 14-45
Test file	<code>fun_with_fi_test.m</code>	MATLAB script that runs both <code>fun_with_matlab</code> and <code>fun_with_fi</code> and compares the results

The `fun_with_fi` Function

The `fun_with_fi` is a fixed-point version of the `fun_with_matlab` function that uses the data types proposed in “Propose Fixed-Point Data Types” on page 14-45.

Variable	Proposed Signedness	Proposed Word Length	Proposed Fraction Length
y	Signed	16	14
x	Signed	16	14
z	Signed	16	15
a	Unsigned	16	18
b	Signed	16	14
i	Unsigned	16	0

For example, in `fun_with_matlab`, variable `y` is defined as `y = zeros(size(x));`. In `fun_with_fi`, to specify that it is a signed fixed-point data type with a word length of 16 and a fraction length of 14:

```
y = fi(zeros(size(x)),1,16,14,'OverflowAction','Wrap','RoundingMethod','Floor');
```

For more information, see `fi`.

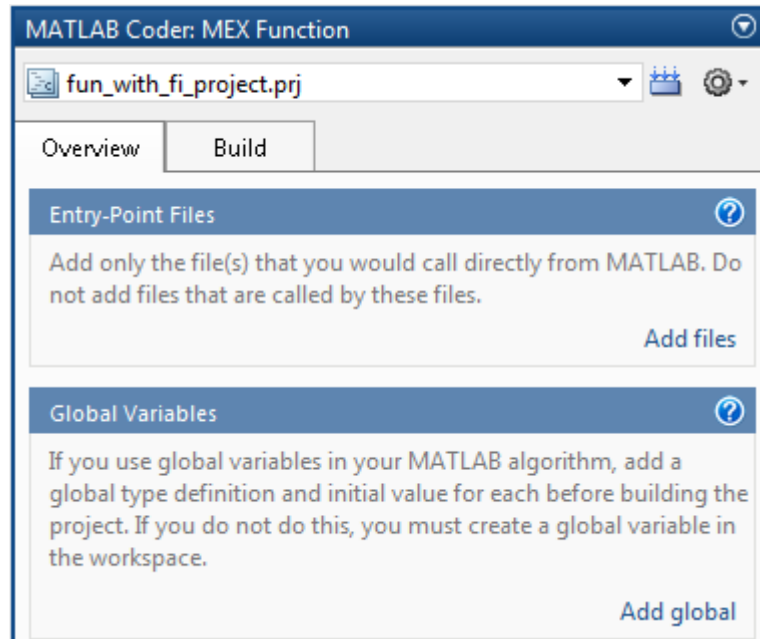
Create and set up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this tutorial.
- 2 On the **MATLAB Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_fi_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_fi_project.prj
```

By default, the project opens in the MATLAB workspace.



- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_fi.m`, and then click **OK** to add the file to the project.

Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `fun_with_fi_test` as a test file, and then click **Run**.

The test file runs and plots the outputs of the filter. MATLAB Coder determines the input types from the test file and then displays them.

- 3 In the Autodefine Input Types dialog box, click **Use These Types** to accept the autodefined input type.

MATLAB Coder sets the type of `x` to `double(1x256)`.

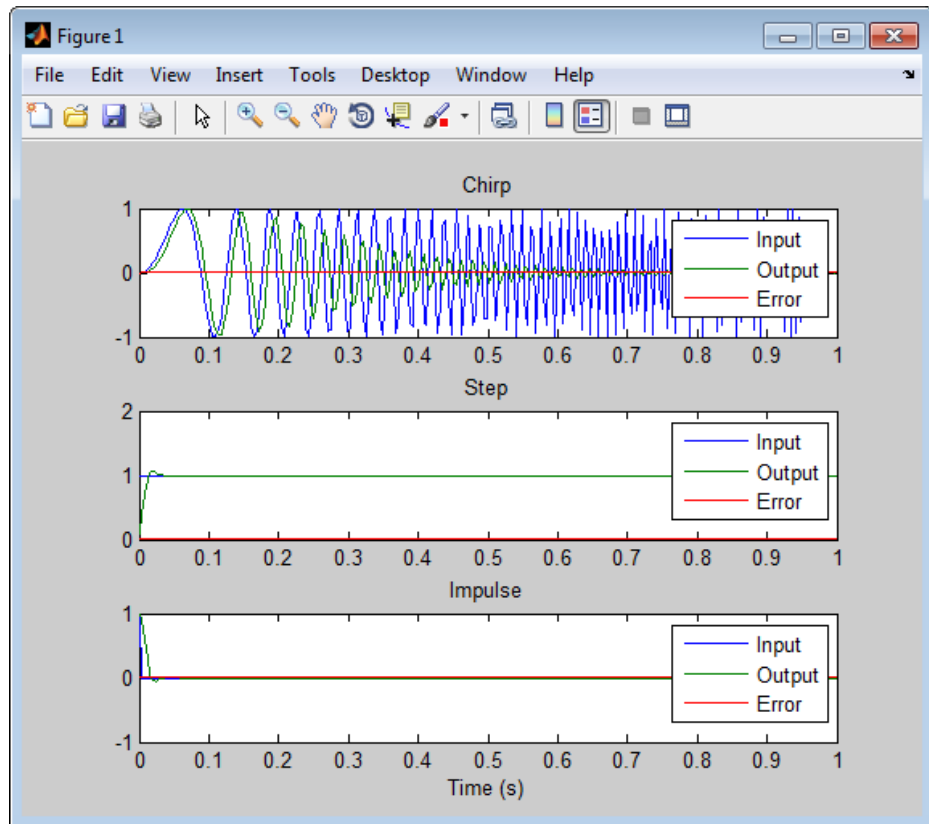
The `fun_with_fi_test` Script

The `fun_with_fi_test` script runs the original floating-point MATLAB algorithm, `fun_with_matlab`, then runs the fixed-point version of the algorithm, `fun_with_fi`. The script then plots the outputs for the floating-point and fixed-point algorithms and the difference in results.

Run Simulation

- 1 In the project, click the **Build** tab.
- 2 On the **Verification** pane, verify that the test file is set to `fun_with_fi_test`. Clear **Redirect entry-point calls to MEX function** so that the test file calls the MATLAB versions of the original and fixed-point algorithms.
- 3 On **Verification** pane, click **Run**.

The `fun_with_fi_test` file runs. The test file runs the original MATLAB algorithm and the fixed-point version, and plots the difference in their outputs.



- 4** Optionally, zoom in on each plot in turn to view the error (difference between the two versions of the algorithm). In this example, the errors are very small, on the order of 10^{-3} . If the error is unacceptably large, refine the fixed-point data types.

Modify Data Type Proposal Settings

When generating instrumented MEX functions, to modify data type proposal settings, on the project **Build** tab, on the **Show Instrumentation Results** pane, click **Data type proposal and report settings**.

Type Proposal Setting	Description
Propose data types	<p>Specify whether to propose data types based on simulation minimum and maximum values. You can view the proposed data types in the code generation report.</p> <p>Dependencies:</p> <ul style="list-style-type: none">• This parameter enables:<ul style="list-style-type: none">▪ Propose fraction lengths for specified word lengths▪ Propose word lengths for specified fraction lengths▪ Default data type of all floating-point expressions▪ Safety margin for min/max values▪ Optimize whole numbers

Type Proposal Setting	Description
<p>Propose fraction lengths for specified word lengths</p>	<p>Select to propose fraction lengths for the word lengths specified in the code.</p> <p>Use simulation minimum and maximum information to propose fraction lengths for variables in your entry-point MATLAB function. MATLAB Coder proposes data types for variables that are scaled doubles and built-in data types only. For floating-point data types in your entry-point function, uses the word length and signedness specified in Default data type of all floating-point expressions to determine the optimal fraction lengths.</p> <p>Dependency:</p> <ul style="list-style-type: none"> • Clearing Propose data types disables this parameter.
<p>Propose word lengths for specified fraction lengths</p>	<p>Select to propose word lengths for the fraction lengths specified in the code.</p> <p>Use simulation minimum and maximum information to propose word lengths for variables in your entry-point MATLAB function. MATLAB Coder proposes data types for variables that are scaled doubles and built-in data types only. For floating-point data types in your entry-point function, uses the fraction length and signedness specified in Default data type of all floating-point expressions to determine the optimal word lengths.</p> <p>Dependency:</p> <ul style="list-style-type: none"> • Clearing Propose data types disables this parameter.

Type Proposal Setting	Description	
Default data type of all floating-point expressions	<p>Specify the default data type to use for floating-point expressions in your entry-point MATLAB function.</p> <p>MATLAB Coder uses this default data type to change the floating-point data types in the code to fixed point.</p> <p>Dependency:</p> <ul style="list-style-type: none"> • Clearing Propose data types disables this parameter. 	
	<code>numerictype([],16,12)</code> (Default)	<p>Set the default data type for floating-point signals to the fixed-point data type specified by <code>numerictype</code>. You can modify the parameters provided to <code>numerictype</code> to specify signedness, word length, and fraction length.</p> <p>Specifying <code>[]</code> for signedness instructs MATLAB Coder to choose the appropriate signedness.</p>
	Remain floating-point	Do not change the data type of floating-point signals.
	<code>int8</code>	Set the default data type for floating-point signals to <code>int8</code> .
	<code>int16</code>	Set the default data type for floating-point signals to <code>int16</code> .
	<code>int32</code>	Set the default data type for floating-point signals to <code>int32</code> .

Type Proposal Setting	Description
Safety margin for min/max values	<p>Specify safety factor for simulation minimum and maximum values.</p> <p>The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of 55 specifies that you want a range at least 55 percent larger. A value of -15 specifies that a range up to 15 percent smaller is acceptable.</p> <p>Dependency:</p> <ul style="list-style-type: none"> • Clearing Propose data types disables this parameter.
Optimize whole numbers	<p>Specify to use integer scaling for variables that were whole numbers during simulation.</p> <p>Dependency:</p> <ul style="list-style-type: none"> • Clearing Propose data types disables this parameter.

Modify Instrumentation Report Settings

When generating instrumented MEX functions, to modify instrumentation report settings, on the project **Build** tab, on the **Show Instrumentation Results** pane, click **Data type proposal and report settings**.

Report Setting	Description
Automatically launch report after running test file	Specify whether to automatically display the report after running the test file.
Open report in a web browser	Specify whether to open the report in a Web browser. Enabling this option allows you to open multiple reports simultaneously.
Create printable report	Specify whether to create a printable report.

Automated Fixed-Point Conversion

In this section...

“License Requirements” on page 14-65

“Fixed-Point Conversion Capabilities” on page 14-65

“Proposing Data Types” on page 14-67

“Viewing Functions” on page 14-68

“Viewing Variables” on page 14-69

“Histogram” on page 14-70

“Function Replacements” on page 14-71

“Validating Types” on page 14-72

“Testing Numerics” on page 14-72

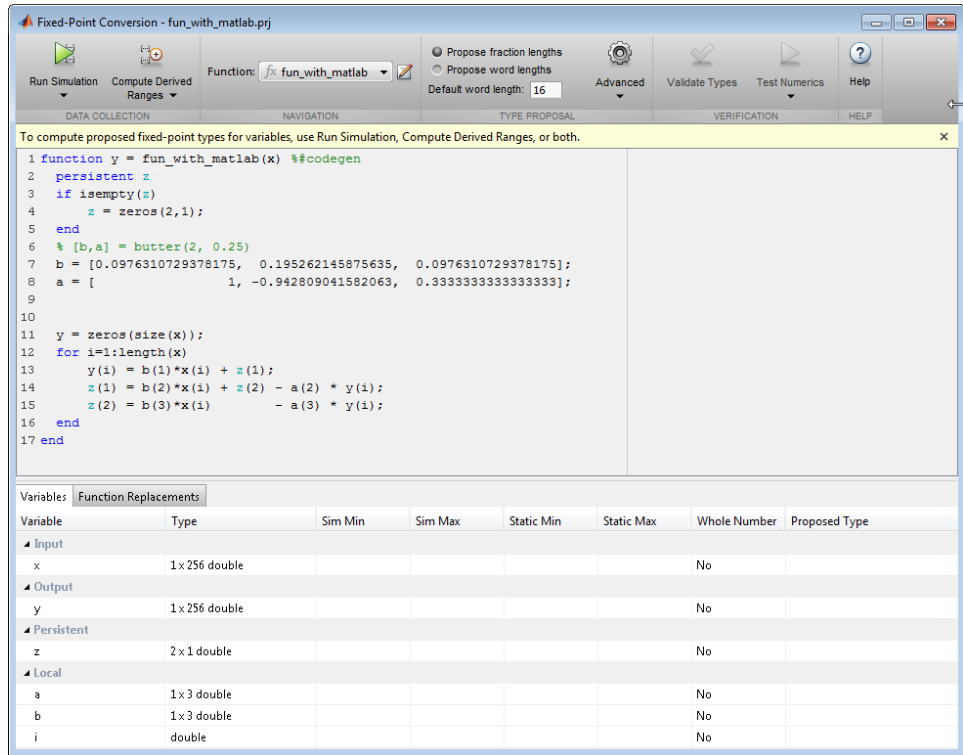
License Requirements

Fixed-point conversion requires the following licenses:

- Fixed-Point Designer
- MATLAB Coder

Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Conversion tool in MATLAB Coder projects. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.



During fixed-point conversion, you can:

- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Validate that you can build your project with the proposed data types.
- Test numerics by running the test file with the fixed-point types applied.
- View a histogram of bits used by each variable.

Fixed-Point Conversion Limitations

- Fixed-point conversion does not support MATLAB classes.
- Derived range analysis is not available on Mac platforms.

Proposing Data Types

The Fixed-Point Conversion tool proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data, or both. If you run a simulation and compute derived ranges, the conversion tool merges the simulation and derived ranges.

Running a Simulation

When you open the Fixed-Point Conversion tool, it generates an instrumented MEX function for your entry-point MATLAB file. If the build completes without errors, the tool displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the tool provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the tool displays them on the **Function Replacements** tab. See “Function Replacements” on page 14-71.

Before running a simulation, specify the test file or files that you want to run. When you run a simulation, the tool runs the test file, calling the instrumented MEX function. If you modify the MATLAB design code, the tool automatically generates an updated MEX function before running a test file.

If the test file runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If the test file fails, the errors are displayed on the **Simulation Output** tab.

Test files should exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test file covers the operating range of the algorithm with the desired accuracy. You can

add test files and select to run more than one test file during the simulation. If you run multiple test files, the conversion tool merges the simulation results.

Optionally, you can select to log histogram data. After running a simulation, you can view the histogram for each variable. For more information, see “Log Histogram Data” on page 14-37.

Computing Derived Ranges

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can promote simulation ranges to use as static ranges. Alternatively, if you know what data type your hardware target uses, set the proposed data type to match this type.

When you select **Compute Derived Ranges**, the tool runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces +/- Inf derived ranges, consider defining ranges for all persistent variables.

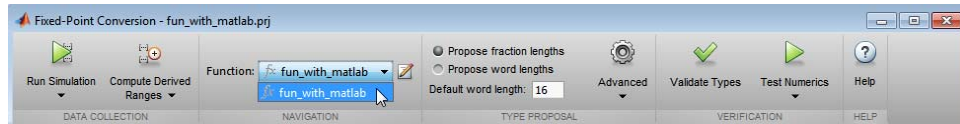
Optionally, you can select **Quick derived range analysis**. With this option, the conversion tool performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. The tool aborts the analysis when the timeout is reached.

Viewing Functions

You can view a list of functions in your project on the **Navigation** pane. This list also includes function specializations. When you select a function from

the list, the MATLAB code for that function is displayed in the Fixed-Point Conversion tool.



Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- **Static Min** and **Static Max** — The static minimum and maximum values.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

When you compute derived ranges, the Fixed-Point Conversion tool runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

The Fixed-Point Conversion tool determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the

tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

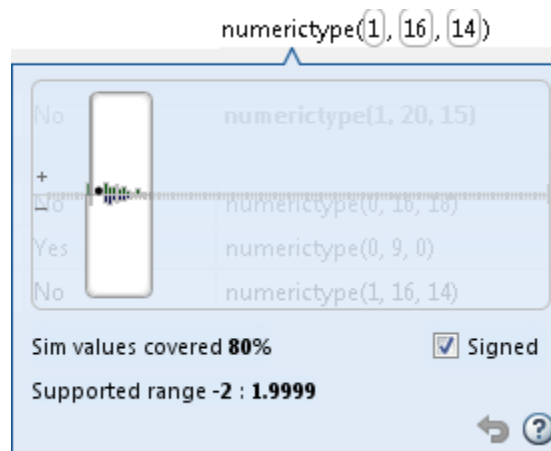
- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

You can also view and edit variable information in the code pane by placing your cursor over a variable name.

You can use `Ctrl+F` to search for variables in the MATLAB code and on the **Variables** tab. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

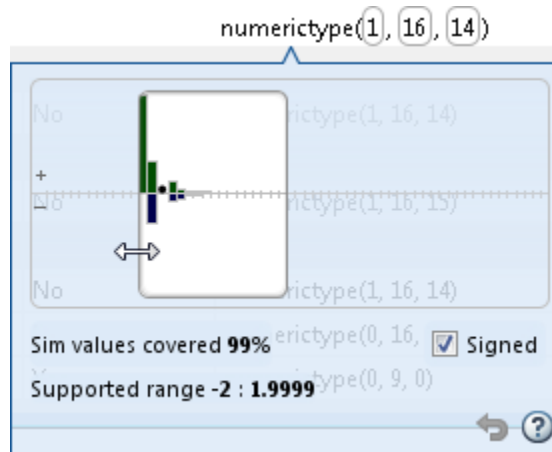
Histogram

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1,16,14)`.




You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click .

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the tool lists these functions on the **Function Replacements** tab. You can add and remove function replacements from this list. If you enter a function replacement for a function, the replacement function is used when you build the project. If you do not enter a replacement, the tool uses the type specified in the original MATLAB code for the function.

Note Using this table, you can replace the names of the functions but you cannot replace argument patterns.

Validating Types

Selecting **Validate Types** validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

Testing Numerics

After validating the proposed fixed-point data types, select **Test Numerics** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test file to define inputs or run a simulation, the tool uses this test file to test numerics. Optionally, you can add test files and select to run more than one test file. The tool compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For non-scalar outputs, only the error information is shown.

By default, the Fixed-Point Conversion tool runs the test files that you added and selected for running the simulation. You can add test files and select to run more than one test file to test numerics.

If the numerical results do not meet your desired accuracy after fixed-point simulation, modify fixed-point data type settings and repeat the type validation and numerical testing steps. You might have to iterate through these steps multiple times to achieve the desired results.

Instrumented MEX Functions

In this section...

“Generating Instrumented MEX Functions” on page 14-73

“Merging Instrumentation Results” on page 14-73

“Clearing Instrumentation Results” on page 14-74

“Redirecting Entry-Point Calls to MEX Function” on page 14-74

“Proposing Fraction Lengths” on page 14-74

“Proposing Word Lengths” on page 14-74

Generating Instrumented MEX Functions

Note This capability is not compatible with automatic fixed-point conversion. If you select Convert to fixed point at build time, you cannot build instrumented MEX functions.

Generating an instrumented MEX function for your MATLAB function enables instrumentation for logging minimum and maximum values of named and intermediate variables in your algorithm. It also enables instrumentation for log2 histograms of named, intermediate and expression values.

When you run the instrumented MEX function, the instrumentation report provides fixed-point data type proposals based on the simulation range data. You can use this information to convert your MATLAB code to fixed point by hand. For more information, see “Propose Fixed-Point Data Types” on page 14-45

Merging Instrumentation Results

When generating instrumented MEX functions, use the **Merge instrumentation results from multiple simulations** option to specify whether to merge new simulation minimum and maximum results with existing simulation results. Merging instrumentation results allows you to collect complete range information from multiple test files.

Clearing Instrumentation Results

When generating instrumented MEX functions, click the **Clear instrumentation results** button to clear instrumentation results from previous runs.

Redirecting Entry-Point Calls to MEX Function

By default, with the **Redirect entry-point calls to MEX function** option selected, the MATLAB Coder software automatically redirects calls to your MATLAB algorithm in the test file to calls to the generated MEX function. The generated MEX function must be in the same folder as the entry-point functions.

If your test file already calls the MEX function, or you want to run the test file to test the original MATLAB algorithm, clear this option.

Proposing Fraction Lengths

When you simulate an instrumented MEX function, if you select to propose fraction lengths for the word lengths specified in the code, MATLAB Coder uses simulation minimum and maximum information and proposes fraction lengths for variables in your entry-point MATLAB function. For floating-point data types in your entry-point function, MATLAB Coder uses the word length and signedness specified in **Default data type of all floating-point expressions** to determine the optimal fraction lengths.

Optionally, specify a safety margin to use when proposing fraction lengths. For more information, see “Modify Data Type Proposal Settings” on page 14-60.

Proposing Word Lengths

When you simulate an instrumented MEX function, if you select to propose word lengths for the fraction lengths specified in the code, MATLAB Coder uses simulation minimum and maximum information and proposes word lengths for variables in your entry-point MATLAB function. For floating-point data types in your entry-point function, MATLAB Coder uses the fraction length and signedness specified in **Default data type of all floating-point expressions** to determine the optimal word lengths.

Optionally, specify a safety margin to use when proposing word lengths. For more information, see “Modify Data Type Proposal Settings” on page 14-60.

Bug Reports

Check Bug Reports for Issues and Fixes

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. Enter the search phrase "Simulation And Code Generation Mismatch" to obtain a report of known bugs where the output of the simulation differs from the output of the generated code.

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.

Setting Up a MATLAB Coder Project

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2
- “Creating a New Project” on page 16-3
- “Opening an Existing Project” on page 16-5
- “Adding Files to the Project” on page 16-6
- “Specifying Properties of Primary Function Inputs in a Project” on page 16-7
- “Autodefine Input Types” on page 16-8
- “Define Input Parameters by Example in a Project” on page 16-12
- “Define or Edit Input Parameter Type in a Project” on page 16-19
- “Define Constant Input Parameters in a Project” on page 16-30
- “Define Inputs Programmatically in the MATLAB File” on page 16-31
- “Adding Global Variables in a Project” on page 16-32
- “Specifying Global Variable Type and Initial Value in a Project” on page 16-33
- “Specify Output File Name” on page 16-40
- “Specify Output File Locations” on page 16-41
- “Selecting Output Type” on page 16-42

MATLAB Coder Project Set Up Workflow

- 1** Create a new project or open an existing one.
- 2** Add the files from which you want to generate code.
- 3** Specify class, size, and complexity of all input parameters.
- 4** Optionally, add global variables.
- 5** Optionally, specify the output file name and output file locations.
- 6** Optionally, select the output type: MEX function (default), Instrumented MEX function, C/C++ static library, C/C++ dynamic library or C/C++ executable.

Creating a New Project

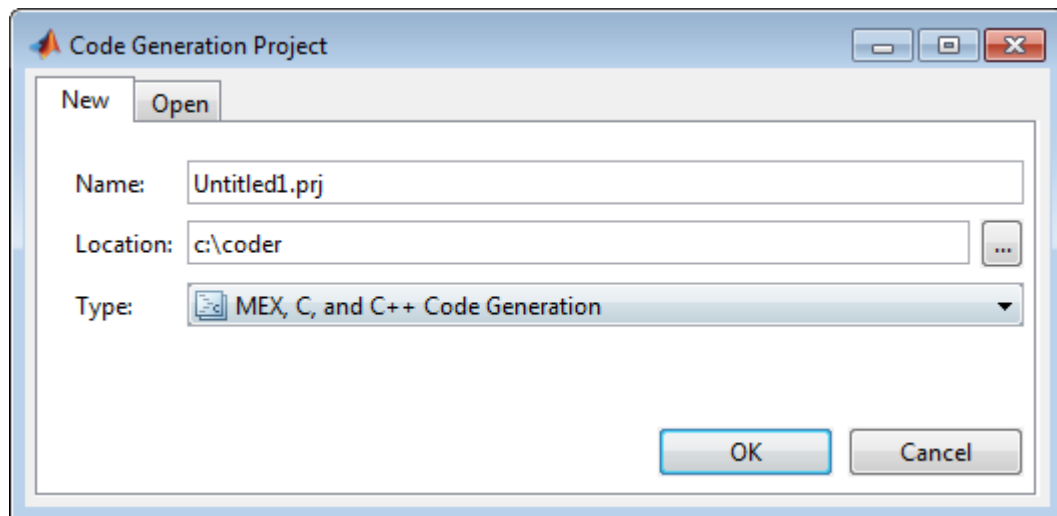
From the MATLAB APPS Tab

- 1 Select the MATLAB **Apps** tab.
- 2 In the **Code Generation** group, click **MATLAB Coder**.
- 3 In the **Code Generation Project** dialog box, on the **New** tab, enter the name of your project in the **Name** field.
- 4 Enter the location of the project in the **Location** field.
Alternatively, use the ... (browse) button to navigate to the location.
- 5 Click **OK**.

At the Command Line

- 1 At the MATLAB command line, enter:

```
coder
```




- 2 In the **Name** field, enter the *project_name*.
- 3 In the **Location** field, enter the location of the project.


Alternatively, use the ... (browse) button to navigate to the location.

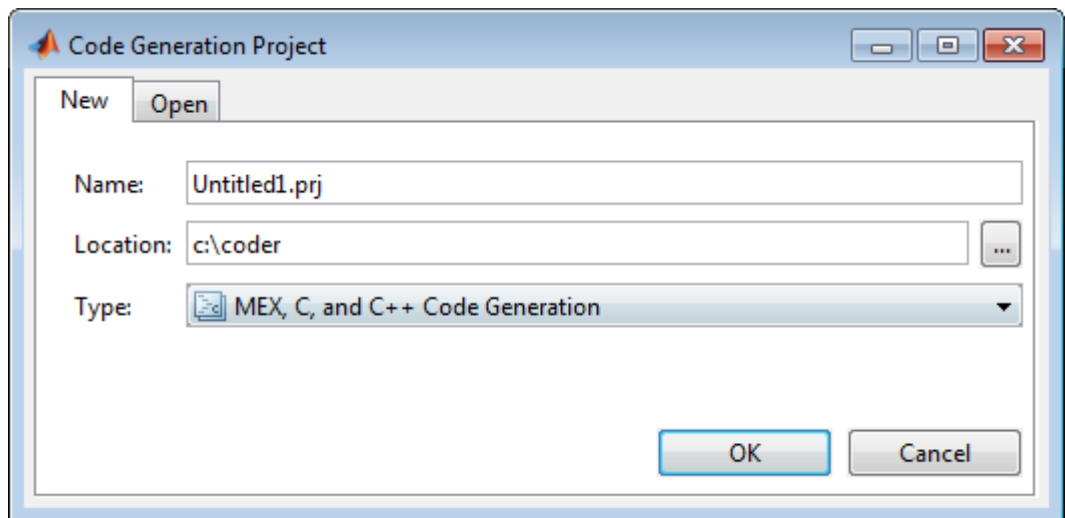
Note The path should not contain spaces, as this can lead to code generation failures in certain operating system configurations. If the path contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not be able to find files on this path.

- 4 Click **OK**.

MATLAB Coder creates a project named *project_name*.prj in the specified location and marks it with the project icon: .

From a MATLAB Coder Project

If you already have a MATLAB Coder project open, in the upper-right corner of the project, click the **Actions** icon () and select **New Project**.



Opening an Existing Project

In this section...

“From the MATLAB APPS Tab” on page 16-5

“At the Command Line” on page 16-5

“From a MATLAB® Coder™ Project” on page 16-5


From the MATLAB APPS Tab

- 1 Select the **MATLAB Apps** tab.
- 2 In the **Code Generation** group, click **MATLAB Coder**.
- 3 In the **Code Generation Project** dialog box, click the **Open** tab.
- 4 From the drop-down list, select a previously opened project or use the **Browse** button to find a project.
- 5 Click **OK**.

At the Command Line

- 1 At the MATLAB command line, enter `coder`.
- 2 In the **Code Generation Project** dialog box, click the **Open** tab.
- 3 From the drop-down list, select a previously opened project or click the **Browse** button to find a project.
- 4 Click **OK**.

From a MATLAB Coder Project

If you already have a MATLAB Coder project open, in the upper-right corner of the project, click the **Actions** icon () and select **Open Project**.

Adding Files to the Project

First, you must add the MATLAB files from which you want to generate code to the project.

- Add only the main (entry-point) files that you call from MATLAB.
- Do not add files that are called by these files.
- Do not add files that have spaces in their names. The path should not contain spaces, as this can lead to code generation failures in certain operating system configurations.
- If the path contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not be able to find files on this path.

To add a file, do one of the following:

- In the project, in the **Entry-Point Files** pane on the **Overview** tab, click the **Add files** link and browse to the file.
- Drag a file from the current folder and drop it in the **Entry-Point Files** pane on the **Overview** tab.

If you add more than one entry-point file, MATLAB Coder lists them alphabetically.

If the functions that you added have inputs, you must define these inputs. See “Specifying Properties of Primary Function Inputs in a Project” on page 16-7.

Specifying Properties of Primary Function Inputs in a Project

Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB Coder must determine the properties of all variables in the MATLAB files at code generation time. To infer variable properties in MATLAB files, MATLAB Coder must be able to identify the properties of the inputs to the entry-point function. Therefore, if your entry-point function has inputs, you must specify the properties of these inputs. If your primary function has no input parameters, MATLAB Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the entry-point function.

You must specify the same number and order of inputs as the MATLAB function unless you use the tilde (~) character to specify unused function inputs. If you use the tilde character, the inputs default to real, scalar doubles.

See Also

- “Properties to Specify” on page 19-38

How to Specify an Input Definition in a Project

Specify an input definition in your MATLAB Coder project using one of the following methods:

- Autodefine Input Types
- Define Type
- Define by Example
- Define Constant
- Define Programmatically in the MATLAB File

Alternatively, specify input definitions at the command line and then use the `codegen` function to generate code. For more information, see “Primary Function Input Specification” on page 19-38.

Autodefine Input Types

In this section...
“How MATLAB Coder Autodefines Input Types” on page 16-8
“Prerequisites for Autodefining Input Types” on page 16-8
“How to Autodefine Input Types” on page 16-8

How MATLAB Coder Autodefines Input Types

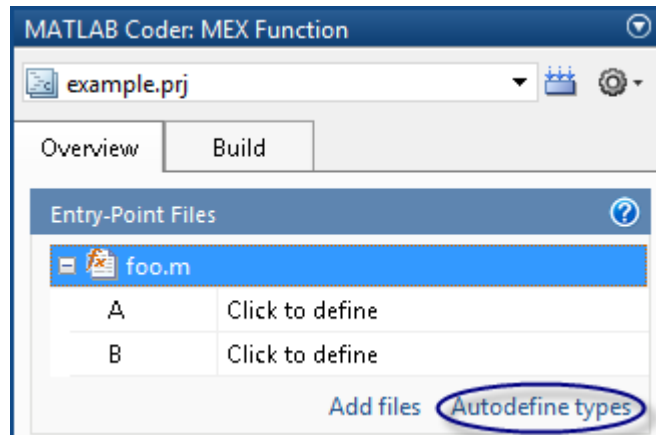
If you specify a test file that calls the project entry-point functions, the MATLAB Coder software can infer the input parameter types by running the test file. If a test file calls an entry-point function multiple times with different sized inputs, the MATLAB Coder software takes the union of the inputs and infers that the inputs are variable size, with an upper bound equal to the size of the largest input.


Prerequisites for Autodefining Input Types

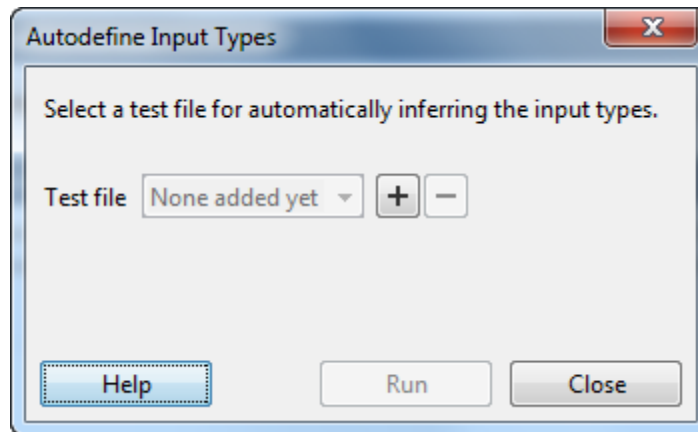
Before using MATLAB Coder to autodefine function input parameter types, you must add at least one entry-point file to your project. You must also specify a test file that calls your entry-point functions with the expected input types. The test file can be either a MATLAB function or a script. It should call the entry-point function at least once.

How to Autodefine Input Types

- 1 On the MATLAB Coder project **Overview** tab, click the **Autodefine types** link.



- 2 In the **Autodefine Input Types** dialog box, click the  button to add a test file to the project.

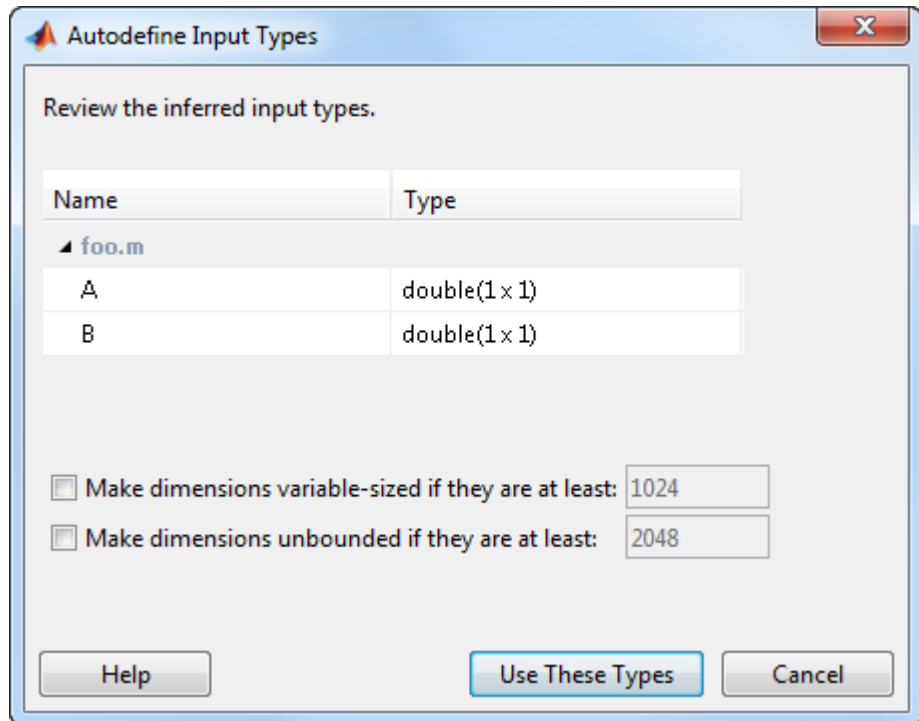


- 3 Browse to the folder that contains the test file and select the file.

Alternatively, if you have already added test files to the project, select one from the list.

- 4 Click the **Run** button.

The software runs the test file and, if the file calls entry-point functions, infers input types for these functions.



The dialog box displays a summary of the inferred types and provides the following options:

- **Make dimensions variable-sized if they are at least**

If you want inputs above a specified size to be variable size with an upper bound, select this option and specify the threshold. If the size, S , of any dimension of an input is equal to or greater than this threshold, the software makes this dimension variable size with an upper bound of S .

- **Make dimensions unbounded if they are at least**

If you want inputs above a specified size to be variable size with no upper bounds (unbounded), select this option and specify the threshold.

If the size of any dimension of an input is equal to or greater than this threshold, the software makes this dimension unbounded.

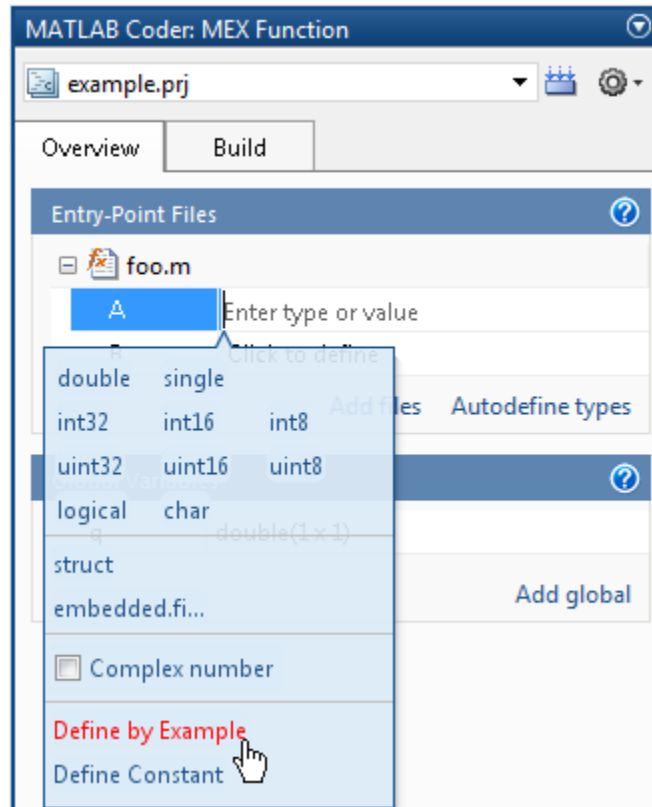
- 5** Review the inferred types. If the types are acceptable, click **Use These Types**. Otherwise, modify your test file, use a different test file to autodefine the types or define them using an alternate method. For more information, see “How to Specify an Input Definition in a Project” on page 16-7.

Define Input Parameters by Example in a Project

In this section...
“How to Define an Input Parameter by Example” on page 16-12
“Specifying Input Parameters by Example” on page 16-13
“Specifying an Enumerated Type Input Parameter by Example” on page 16-15
“Specifying a Fixed-Point Input Parameter by Example” on page 16-17

How to Define an Input Parameter by Example

- 1 On the MATLAB Coder project **Overview** tab, click the input parameter that you want to define.

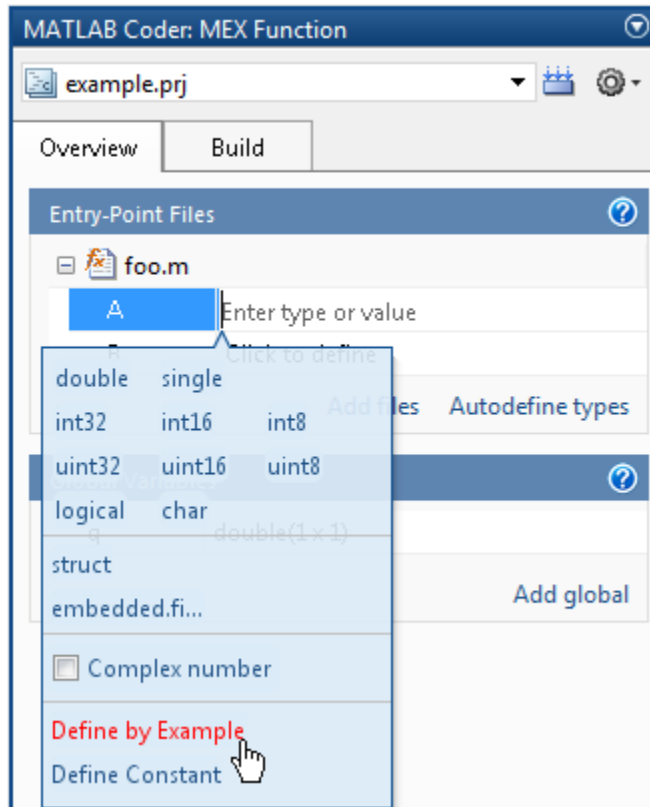


- 2 From the list of input options, select **Define by Example**.
- 3 In the field to the right of the parameter, enter a MATLAB expression. MATLAB Coder software uses the class, size, and complexity of the value of the specified variable or MATLAB expression when compiling the code.

Specifying Input Parameters by Example

This example shows how to specify a 1-by-4 vector of unsigned 16-bit integers.

- 1 On the MATLAB Coder project **Overview** tab, click the input parameter that you want to define.



2 From the list of input options, select Define by Example.

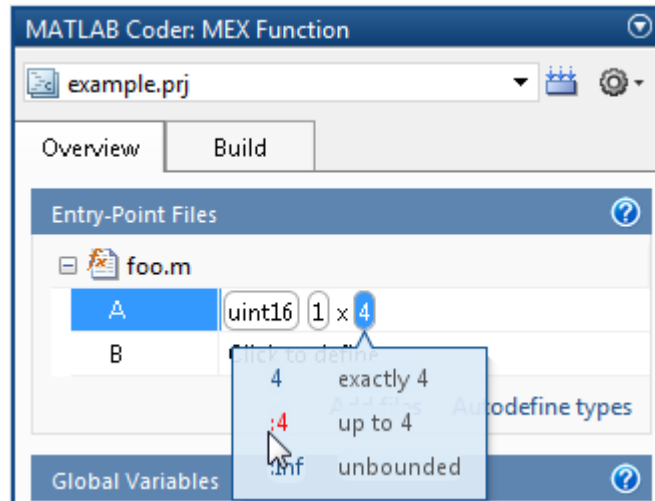
3 In the field to the right of the parameter, enter:

```
zeros(1,4, 'uint16')
```

The input type is uint16(1x4).

4 Optionally, after specifying the input type, you can specify that the input is variable size.

Select the second dimension.



- From the list of size options, select `:4` to specify that the second dimension is variable size with an upper bound of 4. Alternatively, select `:Inf` to specify that the second dimension is unbounded.

Alternatively, you can specify that the input is variable size by using the `coder.newtype` function. Enter the following MATLAB expression:

```
coder.newtype('uint16',[1 4],[0 1])
```

Note To specify that an input is a double-precision scalar, simply enter 0.

Specifying an Enumerated Type Input Parameter by Example

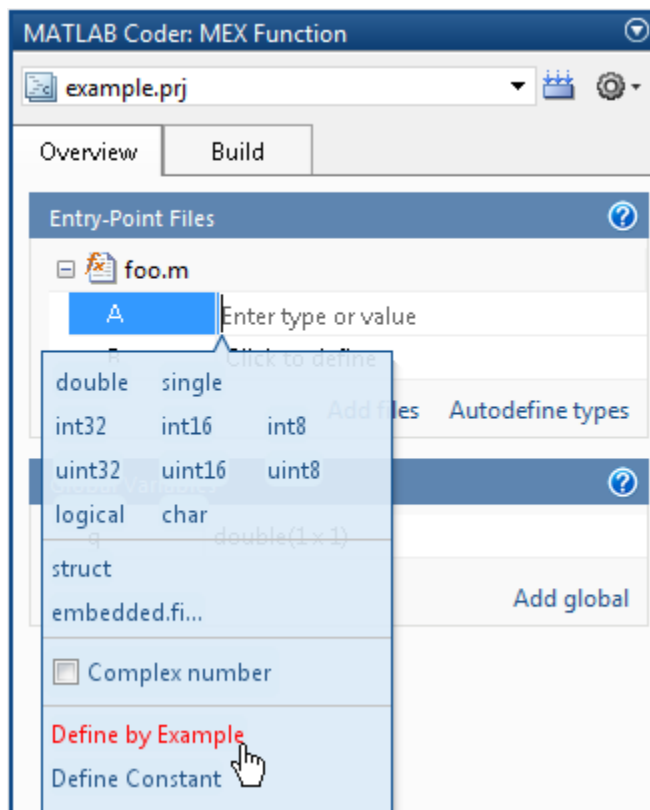
This example shows how to specify that an input uses the enumerated type `MyColors`.

- Define an enumeration `MyColors`. On the MATLAB path, create a file named `'MyColors'` containing:

```
classdef(Enumeration) MyColors < int32
    enumeration
```

```
        green(1),  
        red(2),  
    end  
end
```

- 2 On the MATLAB Coder project **Overview** tab, click to the input parameter that you want to define.



- 3 From the list of input options, select Define by Example.
- 4 In the field to the right of the parameter, enter the following MATLAB expression:

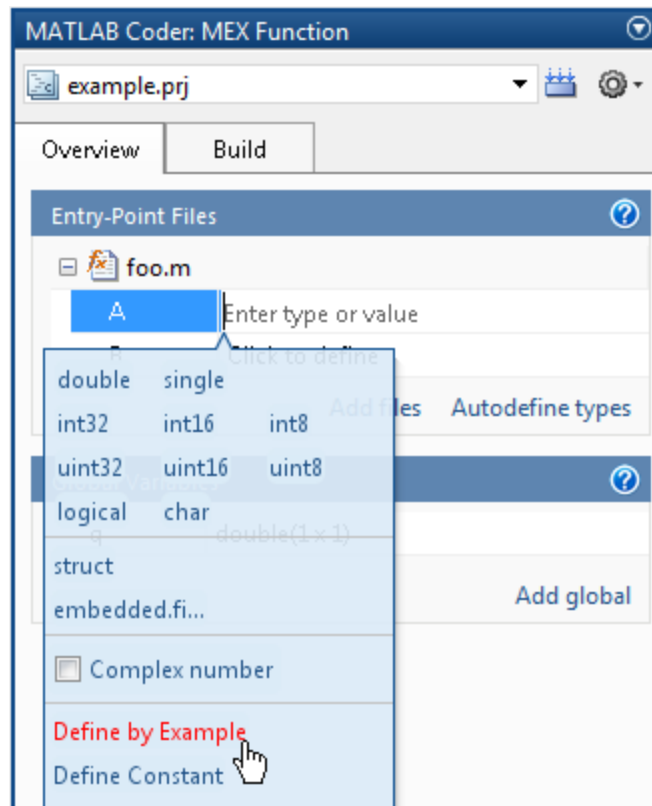
MyColors.red

Specifying a Fixed-Point Input Parameter by Example

To specify fixed-point inputs, you must install Fixed-Point Designer software.

This example shows how to specify a signed fixed-point type with a word length of 8 bits, and a fraction length of 3 bits.

- 1 On the MATLAB Coder project **Overview** tab, click the input parameter that you want to define.



- 2 From the list of input options, select **Define by Example**.

3 In the field to the right of the parameter, enter:

```
fi(10, 1, 8, 3)
```

MATLAB Coder sets the type of input `u` to `embedded.fi(1x1)`. By default, if you have not specified a local `fi`math, MATLAB Coder uses the default `fi`math. For more information, see “`fi`math for Sharing Arithmetic Rules”.

Optionally, modify the fixed-point properties of the input, see “Specifying a Fixed-Point Input Parameter by Type” on page 16-21 or the size of the input, see “Define or Edit Input Parameter Type in a Project” on page 16-19.

Define or Edit Input Parameter Type in a Project

In this section...
“How to Define or Edit an Input Parameter Type” on page 16-19
“Specifying an Enumerated Type Input Parameter by Type” on page 16-21
“Specifying a Fixed-Point Input Parameter by Type” on page 16-21
“Specifying Structures” on page 16-23

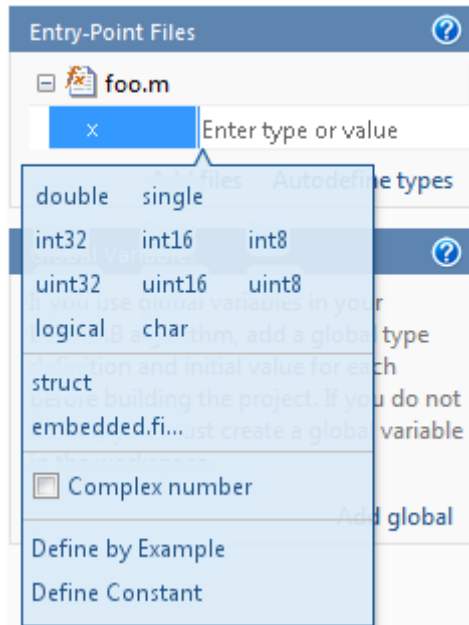
How to Define or Edit an Input Parameter Type

The following procedure is for input types `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, `logical`, and `char`.

For more information about defining other types, see the following table.

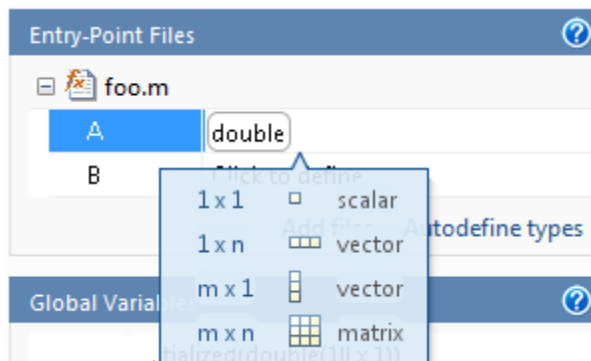
Input Type	Link
A structure (<code>struct</code>)	“Specifying Structures” on page 16-23
A fixed-point data type (<code>embedded.fi</code>)	“Specifying a Fixed-Point Input Parameter by Type” on page 16-21
An input by example (Define by Example)	“Define Input Parameters by Example in a Project” on page 16-12
A constant (Define Constant)	“Define Constant Input Parameters in a Project” on page 16-30

- 1 On the **Overview** tab **Entry-Point Files** pane, click the field to the right of the input parameter name to view the input options.



- 2 Optionally, for numeric types, select **Complex number** to make the parameter a complex type. By default, inputs are real.
- 3 Select the input type.

The selected type is displayed for the input parameter together with size options.



- 4** From the list, select whether your input is a scalar, a $1 \times n$ vector, a $m \times 1$ vector or a $m \times n$ matrix. By default, if you do not select a size option, MATLAB Coder defines inputs as scalars.
- 5** Optionally, if your input is not scalar, enter sizes m and n . You can specify:
 - Fixed size, for example, 10.
 - Variable size, up to a specified limit, by using the `:` prefix. For example, to specify that your input can vary in size up to 10, enter `:10`.
 - Unbounded variable size by entering `:Inf`.

You can edit the size of each dimension after specifying it.

Specifying an Enumerated Type Input Parameter by Type

To specify that an input uses the enumerated type `MyColors`:

- 1** Define an enumeration `MyColors`. On the MATLAB path, create a file named `'MyColors'` containing:

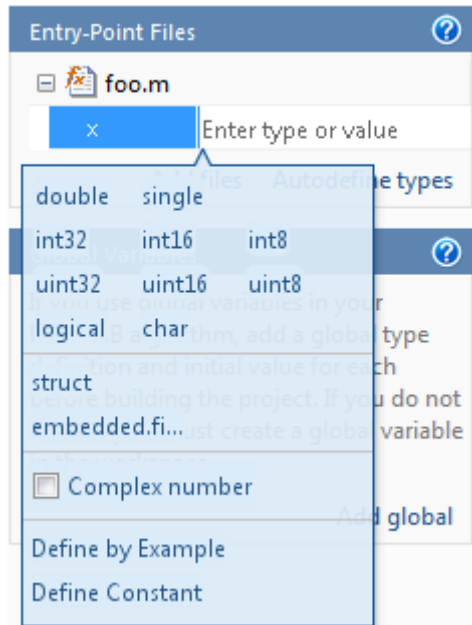
```
classdef(Enumeration) MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

- 2** In the field to the right of the input parameter, enter `MyColors`.

Specifying a Fixed-Point Input Parameter by Type

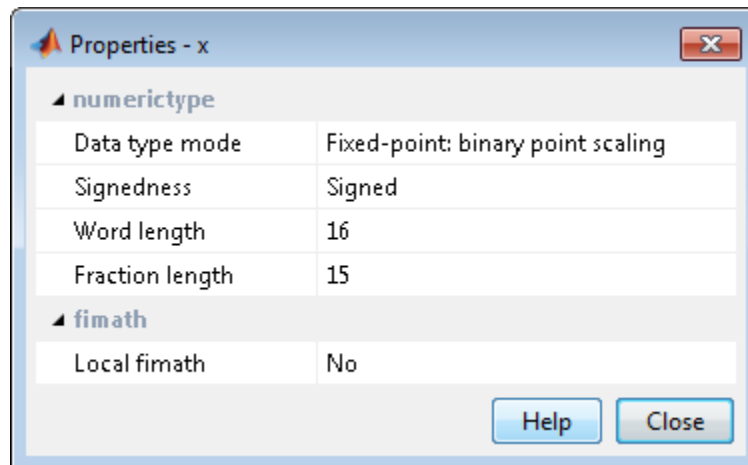
To specify fixed-point inputs, you must install Fixed-Point Designer software.

- 1** On the **Overview** tab **Entry-Point Files** pane, click the box to the right of the input parameter name to view the input options.



- 2 Select `embedded.fi...`.

The **Properties** dialog box opens.



- 3 In this dialog box, set up the input parameter `numericType` and `fimath` properties and then close the dialog box.

If you do not specify a local `fimath`, MATLAB Coder uses the default `fimath`. For more information, see “Default `fimath` Usage to Share Arithmetic Rules”.

- 4 The size of the input defaults to `1x1`. Optionally, modify the size by selecting the dimension that you want to change and entering a new size.

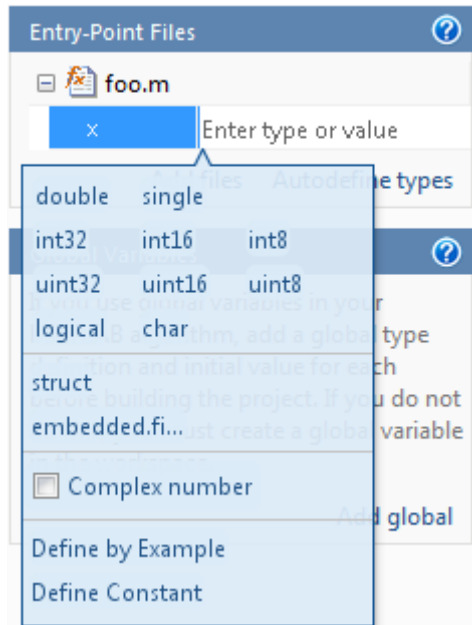
Specifying Structures

When a primary input is a structure, MATLAB Coder treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition, as follows:

- For each field of input structures, specify class, size, and complexity.
- For each field that is fixed-point class, also specify `numericType`, and `fimath`.

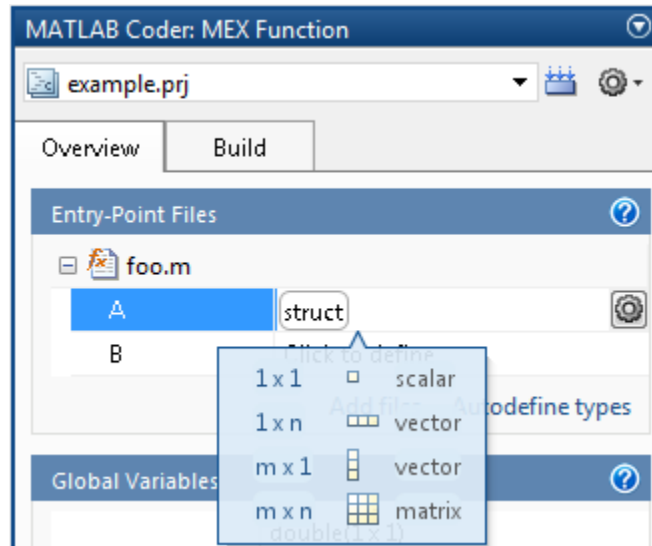
Specifying Structures by Type

- 1 On the **Overview** tab **Entry-Point Files** pane, click the field to the right of the input parameter name to view the input options.



- 2 From the list of input options, select struct.

The selected type, struct, is displayed for the input parameter together with size options.

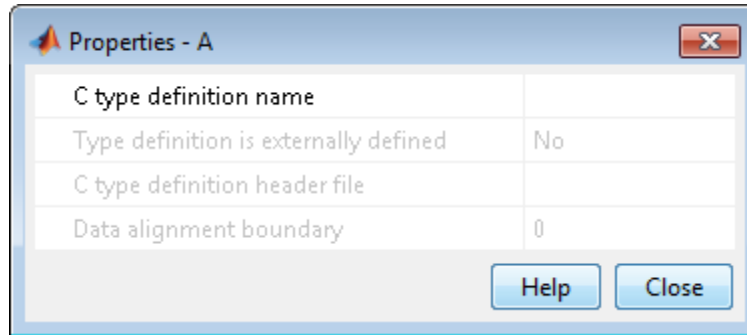


- 3 From the list, select whether your structure is a scalar, $1 \times n$ vector, $m \times 1$ vector or $m \times n$ matrix. By default, if you do not select a size option, MATLAB Coder defines inputs as scalars.
- 4 Optionally, if your input is not scalar, enter sizes m and n . You can specify:
 - Fixed size, for example, 10.
 - Variable size, up to a specified limit, by using the `:` prefix. For example, to specify that your input can vary in size up to 10, enter `:10`.
 - Unbounded variable size by entering `:Inf`.
- 5 Optionally, add fields to the structure as described in “How to Add a Field to a Structure” on page 16-28 and then set their size and complexity.
- 6 Optionally, specify properties for the structure in the generated code as described in “How to Set Structure Properties” on page 16-25.

How to Set Structure Properties

- 1 To the right of the structure definition, click the **Actions** icon, (⚙).

The structure properties dialog box opens.



2 Specify properties for the structure in the generated code.

Property	Description
C type definition name	Name to use for the structure variable in the generated code.
Type definition is externally defined	<p>Default: No — type definition is not externally defined</p> <p>If you select 'Yes' to declare an externally defined structure, MATLAB Coder does not generate the definition of the structure type; you must provide it in a custom include file.</p> <p>Dependency: This option is enabled by C type definition name.</p>

Property	Description
C type definition header file	<p>Name of the header file that contains the external definition of the structure, for example, "mystruct.h". Specify the path to the file using the Additional include directories parameter on the Project Settings dialog box Custom Code tab.</p> <p>By default, the generated code contains #include statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the C type definition header file option, MATLAB Coder includes that header file exactly at the point where it is required.</p> <p>Must be a non-empty string.</p> <p>Dependency: This option is enabled when Type definition is externally defined is set to Yes.</p>
Data alignment boundary	<p>The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned</p>

Property	Description
	<p>on any specific boundary so it will not be matched by CRL functions that require alignment.</p> <p>Alignment must be either -1 or a power of 2 that is no more than 128.</p> <p>Default: 0</p> <p>Dependency: This option is enabled when Type definition is externally defined is set to Yes.</p>

How to Rename a Field in a Structure

On the project **Overview** tab, select the name field of the structure that you want to rename and enter the new name.

How to Add a Field to a Structure


- 1 On the project **Overview** tab, select the structure to which you want to add a field.
- 2 To the right of the structure, click the **Actions** icon (⚙️) to open the context menu.
- 3 From the menu, select **Add Field**.

If the structure already contains other fields, MATLAB Coder adds the field after the existing fields.


- 4 Enter the field name and define its type.

How to Insert a Field into a Structure

- 1 On the project **Overview** tab, select the field under which you want to add another field.

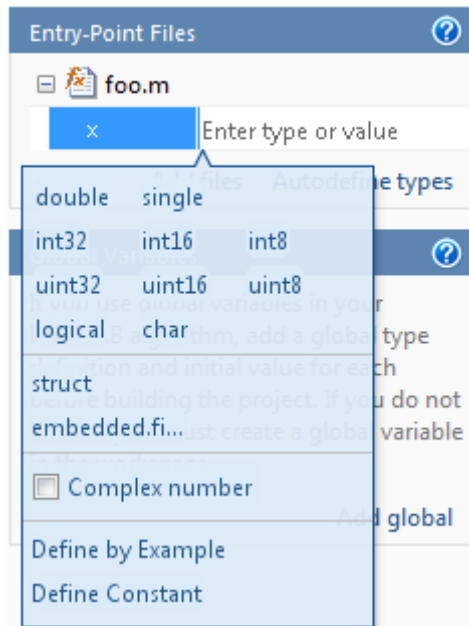
- 2** To the right of the structure, click the **Actions** icon () to open the context menu.
- 3** From the menu, select **Insert Field**.
MATLAB Coder adds the field after the selected field.
- 4** Enter the field name and define its type.

How to Remove a Field from a Structure

- 1** In the project **Overview** tab, select the field that you want to remove.
- 2** To the right of the structure, click the **Actions** icon () to open the context menu.
- 3** From the menu, select **Remove Field**.

Define Constant Input Parameters in a Project

- 1 On the **Overview** tab **Entry-Point Files** pane, click the field to the right of the input parameter name to view the input options.



- 2 Select **Define Constant**.
- 3 In the field to the right of the parameter name, enter the value of the constant or a MATLAB expression that represents the constant.

MATLAB Coder software uses the value of the specified MATLAB expression as a compile-time constant.

Define Inputs Programmatically in the MATLAB File

You can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB entry-point files.

To enable this option, on the **Project Settings** dialog box **Advanced** pane, select **Determine input types from source code preconditions**. If you enable this option:

- MATLAB Coder labels all entry-point function inputs as `Deferred` and determines the input types at compile time.
- You cannot specify input types in this project using any other input specification method.

For more information, see “Define Input Properties Programmatically in the MATLAB File” on page 19-50.

Adding Global Variables in a Project

To add global variables to the project:

- 1** On the project **Overview** tab, click **Add global**.

By default, MATLAB Coder names the first global variable in a project **g**, and subsequent global variables **g1**, **g2**, etc.

- 2** Enter the name of the global variable.
- 3** After adding a global variable, before building the project, specify its type and initial value. If you do not do this, you must create a variable with the same name in the global workspace. See “Specifying Global Variable Type and Initial Value in a Project” on page 16-33.

Specifying Global Variable Type and Initial Value in a Project

In this section...

“Why Specify a Type Definition for Global Variables?” on page 16-33

“How to Specify a Global Variable Type” on page 16-33

“Defining a Global Variable by Example” on page 16-34

“Defining or Editing Global Variable Type” on page 16-35

“Defining Global Variable Initial Value” on page 16-37

“Removing Global Variables” on page 16-39

Why Specify a Type Definition for Global Variables?

If you use global variables in your MATLAB algorithm, before building the project, you must add a global type definition and initial value for each. If you do not initialize the global data, MATLAB Coder looks for the variable in the MATLAB global workspace. If the variable does not exist, MATLAB Coder generates an error.

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable. At code generation time, MATLAB Coder needs to have an initial value to determine the type of a global variable. Otherwise, the global variable might be used before it is defined and then MATLAB Coder cannot determine the type to use in the generated code.

For MEX functions, if you use global data, you must also specify whether to synchronize this data between MATLAB and the MEX function. For more information, see “Synchronizing Global Data with MATLAB” on page 19-83.

How to Specify a Global Variable Type

1 Specify the type of each global variable using one of the following methods:

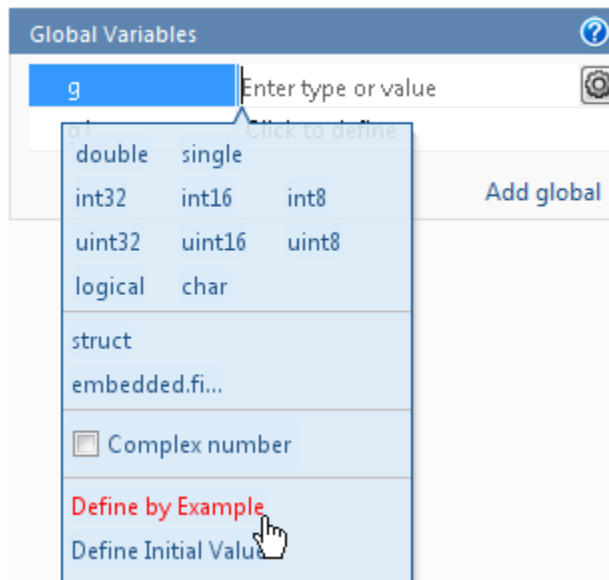
- Define by example
- Define type

- 2 Define an initial value for each global variable.

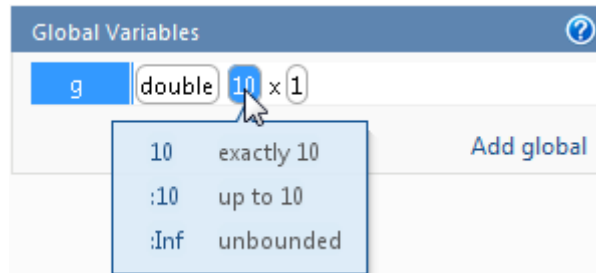
If you do not provide a type definition and initial value for a global variable, you **must** create a variable with the same name and suitable class, size, complexity, and value in the MATLAB workspace.

Defining a Global Variable by Example

- 1 On the project **Overview** tab, click the field to the right of the global variable that you want to define.



- 2 Select **Define by Example**.
- 3 In the field to the right of the global name, enter a MATLAB expression that has the required class, size, and complexity. MATLAB Coder software uses the class, size, and complexity of the value of this expression as the type for the global variable.
- 4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, 10.



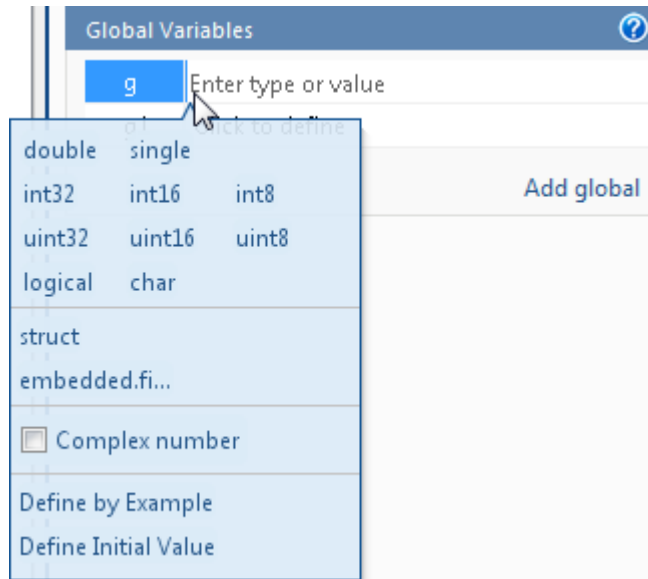
You can specify:

- Fixed size. In this example, select 10.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to 10, select `:10`.
- Unbounded variable size by selecting `:Inf`.

Note You define global variables in the same way that you define input parameters. For more information, see “Define Input Parameters by Example in a Project” on page 16-12

Defining or Editing Global Variable Type

- 1 On the project **Overview** tab, click the field to the right of the global variable that you want to define.

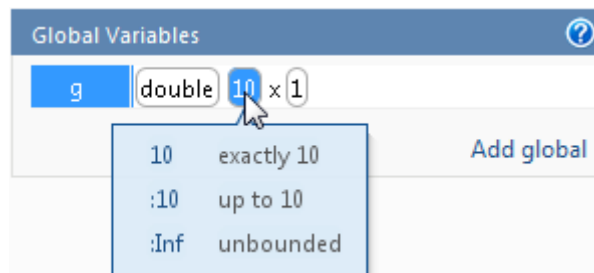


2 Optionally, for numeric types, select **Complex** to make the parameter a complex type. By default, inputs are real.

3 Select the type for the global variable. For example, double.

By default, the global variable is a scalar.

4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, 10.



You can specify:

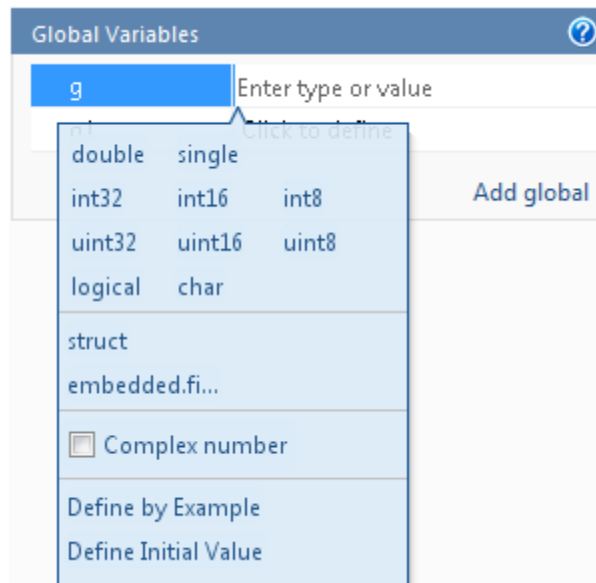
- Fixed size. In this example, select 10.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to 10, select `:10`.
- Unbounded variable size by selecting `:Inf`.

Defining Global Variable Initial Value

- “Define Initial Value Before Defining Type” on page 16-37
- “Define Initial Value After Defining Type” on page 16-38

Define Initial Value Before Defining Type

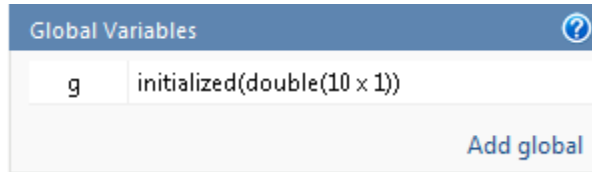
- 1 On the project **Overview** tab, click the field to the right of the global variable.



- 2 Select **Define Initial Value**.
- 3 Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

Because you did not define the type of the global variable before you defined its initial value, MATLAB Coder uses the type of the initial value as the global variable type.

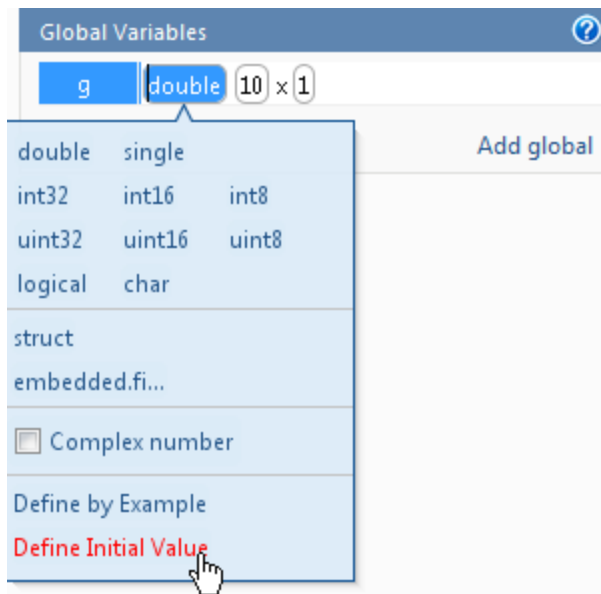
The project displays that the global variable is initialized.



If you change the type of a global variable after defining its initial value, you must redefine the initial value.

Define Initial Value After Defining Type

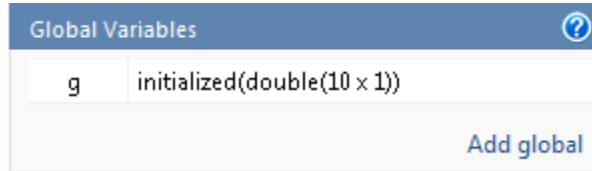
- 1 On the project **Overview** tab, click the type field of the global variable.



- 2 Select **Define Initial Value**.

- 3 Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

The project displays that the global variable is initialized.



Removing Global Variables

- 1 On the project **Overview** tab, select the global variable that you want to remove.
- 2 To the right of the variable, click the **Actions** icon (⚙) to open the context menu.
- 3 From this menu, select **Remove Global**.

MATLAB Coder removes the global variable.

Specify Output File Name

On the project **Build** tab, in the **Output file** field, enter the file name. The file name can include an existing path.

Note Do not put any spaces in the file name.

By default, if the name of the first entry-point MATLAB file is *fcn1*, the output file name is:

- *fcn1* for C/C++ libraries and executables.
- *fcn1_mex* for MEX functions.

By default, MATLAB Coder generates files in the folder *project_folder/codegen/target/fcn1*:

- *project_folder* is your current project folder
- target is:
 - *mex* for MEX functions
 - *lib* for static C/C++ libraries
 - *dll* for dynamic C/C++ libraries
 - *exe* for C/C++ executables

To learn how to change the default output folder, see “Specify Output File Locations” on page 16-41.

Command Line Alternative

Use the `codegen` function `-o` option.

Specify Output File Locations

The path should not contain:

- Spaces, as this can lead to code generation failures in certain operating system configurations.
- Non 7-bit ASCII characters, such as Japanese characters.

1 On the project **Build** tab, click **More settings**.

2 In the Project Settings dialog box, click the **Paths** tab.

The default setting for the **Build Folder** field is `A subfolder of the project folder`. By default, MATLAB Coder generates files in the folder `project_folder/codegen/target/fcn1`:

- `fcn1` is the name of the first entry-point file
- `target` is:
 - `mex` for MEX functions
 - `dll` for dynamic C/C++ libraries
 - `lib` for static C/C++ libraries
 - `exe` for C/C++ executables

3 To change the output location, you can either:

- Set **Build Folder** to `A subfolder of the current MATLAB working folder`

MATLAB Coder generates files in the `MATLAB_working_folder/codegen/target/fcn1` folder

- Set **Build Folder** to `Specified folder`. In the **Build folder name** field, provide the path to the folder.

Command Line Alternative

Use the `codegen` function `-d` option.

Selecting Output Type

On the project **Build** tab, from the **Output type** drop-down list, select one of the available output types:

- MEX Function (default)
- Instrumented MEX Function

Building an instrumented MEX function requires a Fixed-Point Designer license and clears prior instrumentation results.

- C/C++ Static Library
- C/C++ Dynamic Library
- C/C++ Executable

Command Line Alternative

Use the `codegen` function `-config` option.

Changing Output Type

MEX functions use a different set of configuration parameters than C/C++ libraries and executables use. When you switch the output type between MEX Function or Instrumented MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, you should verify these settings.

If you enable any of the following parameters when the output type is MEX Function or Instrumented MEX Function, and you want to use the same setting for C/C++ code generation as well, you must enable it again for C/C++ Static Library, C/C++ Dynamic Library, and C/C++ Executable.

Check These MATLAB Coder Project Parameters When Changing Output Type

Project Settings Dialog Box Tab	Parameter Name
Paths	Working folder
	Build folder
	Search paths
Speed	Saturate on integer overflow
Memory	Enable variable-sizing
	Dynamic memory allocation
	Stack usage max
Code Appearance	Generated file partitioning method
	Include comments
	MATLAB source code as comments
	Reserved names
Debugging	Always create a code generation report
	Automatically launch a report if one is generated
Custom Code	Source file
	Header file
	Initialize function
	Terminate function
	Additional include directories
	Additional source files
	Additional libraries
	Post-code-generation command

Project Settings Dialog Box Tab	Parameter Name
Advanced	Constant folding timeout
	Language
	Inline threshold
	Inline threshold max
	Inline stack limit
	Use memcpy for vector assignment
	Memcpy threshold (bytes)
	Use memset to initialize floats and doubles to 0.0

Check These Command-Line Parameters When Changing Output Type

When you switch between MEX and C output types, check these `coder.MexCodeConfig`, `coder.CodeConfig` or `coder.EmbeddedCodeConfig` configuration object parameters, as applicable.

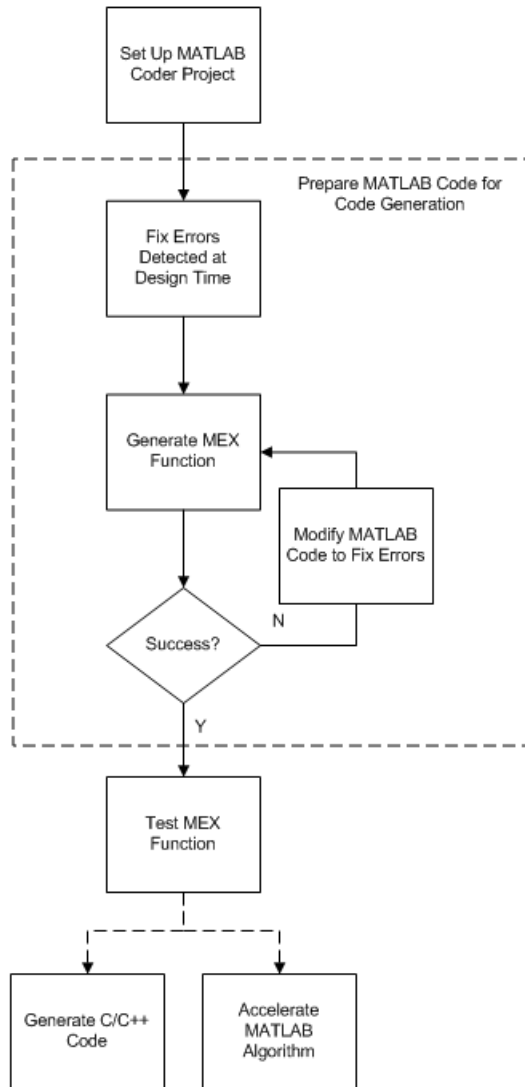
- `ConstantFoldingTimeout`
- `CustomHeaderCode`
- `CustomInclude`
- `CustomInitializer`
- `CustomLibrary`
- `CustomSource`
- `CustomSourceCode`
- `CustomTerminator`
- `DynamicMemoryAllocation`
- `EnableMemcpy`
- `EnableVariableSizing`

- FilePartitionMethod
- GenCodeOnly
- GenerateComments
- GenerateReport
- InitFltsAndDblsToZero
- InlineStackLimit
- InlineThreshold
- InlineThresholdMax
- LaunchReport
- MATLABSourceComments
- MemcpyThreshold
- PostCodeGenCommand
- ReservedNameArray
- SaturateOnIntegerOverflow
- StackUsageMax
- TargetLang

Preparing MATLAB Code for C/C++ Code Generation

- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Fixing Errors Detected at Design Time” on page 17-4
- “Using the Code Analyzer” on page 17-5
- “Check Code With the Code Analyzer” on page 17-6
- “Check Code Using the Code Generation Readiness Tool” on page 17-8
- “Code Generation Readiness Tool” on page 17-10
- “Unable to Determine Code Generation Readiness” on page 17-16
- “Generate MEX Functions Using the MATLAB® Coder™ Project Interface” on page 17-17
- “Generate MEX Functions at the Command Line” on page 17-25
- “Fix Errors Detected at Code Generation Time” on page 17-27
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 17-28
- “Running MEX Functions” on page 17-30
- “Debugging Strategies” on page 17-31

Workflow for Preparing MATLAB Code for Code Generation



See Also

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2
- “Fixing Errors Detected at Design Time” on page 17-4
- “Generate MEX Functions Using the MATLAB® Coder™ Project Interface” on page 17-17
- “Fix Errors Detected at Code Generation Time” on page 17-27
- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “C/C++ Code Generation” on page 19-5
- “Accelerate MATLAB Algorithms” on page 22-11

Fixing Errors Detected at Design Time

Use the code analyzer and the code generation readiness tool to detect issues at design time. Before generating code, you must fix these issues.

See Also

- “Check Code With the Code Analyzer” on page 17-6
- “Check Code Using the Code Generation Readiness Tool” on page 17-8
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 17-28
- “Debugging Strategies” on page 17-31

Using the Code Analyzer

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing compilation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications.

To use the code analyzer to identify warnings and errors specific to MATLAB for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of code generation analyzer messages is available in the MATLAB Code Analyzer preferences. For more information, see “Running the Code Analyzer Report”.

Note The code analyzer might not detect all MATLAB for code generation issues. After eliminating the errors or warnings that the code analyzer detects, compile your code with MATLAB Coder to determine if the code has other compliance issues.

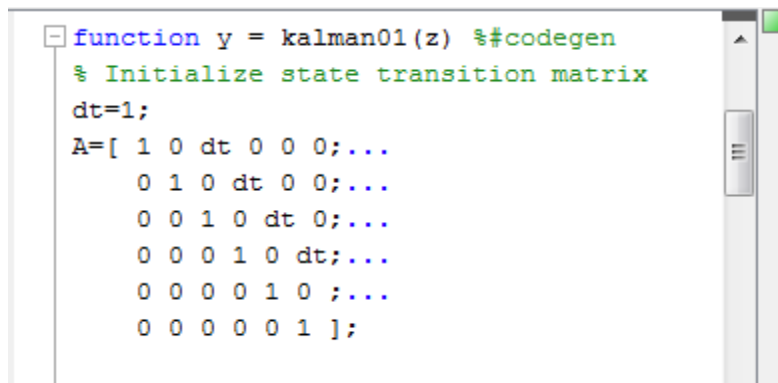
Check Code With the Code Analyzer

The code analyzer checks your code for problems and recommends modifications. You can use the code analyzer to check your code interactively in the MATLAB Editor while you work.

To verify that continuous code checking is enabled:

- 1 In MATLAB, select the **Home** tab and then click **Preferences**.
- 2 In the **Preferences** dialog box, select **Code Analyzer**.
- 3 In the **Code Analyzer Preferences** pane, verify that **Enable integrated warning and error messages** is selected.

The code analyzer provides an indicator in the top right of the editor window. If the indicator is green, the analyzer did not detect code generation issues.



```
function y = kalman01(z) %#codegen
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

If the indicator is red, the analyzer has detected errors in your code. If it is orange, it has detected warning. When the indicator is red or orange, a red or orange marker appears to the right of the code where the error occurs. Place your pointer over the marker for information about the error. Click the underlined text in the error message for a more detailed explanation and suggested actions to fix the error.

```
p_prd = A * p_est * A' + Q;

% Estimation
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';

% Estimated state and covariance
x_est = x_prd + klm_gain * (z(1:2,i) - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;

% Compute the estimated measurements
v(:,i) = H * x_est;
end
end
```

✖ Line 46: [Code generation requires variable 'y' to be fully defined before subscripting it.](#)
✖ Line 46: [Code generation does not support variable 'y' size growth through indexing.](#)

Before generating code from your MATLAB code, you must fix the errors detected by the code analyzer.

Check Code Using the Code Generation Readiness Tool

In this section...
“Run Code Generation Readiness Tool at the Command Line” on page 17-8
“Run Code Generation Readiness Tool from the Current Folder Browser” on page 17-8
“Run the Code Generation Readiness Tool in a Project” on page 17-9
“See Also” on page 17-9

Run Code Generation Readiness Tool at the Command Line

1 Navigate to the folder that contains the file that you want to check for code generation readiness.

2 At the MATLAB command prompt, enter:

```
coder.screener('filename')
```

The **Code Generation Readiness** tool opens for the file named `filename`, provides a code generation readiness score, and lists issues that must be fixed prior to code generation.

Run Code Generation Readiness Tool from the Current Folder Browser

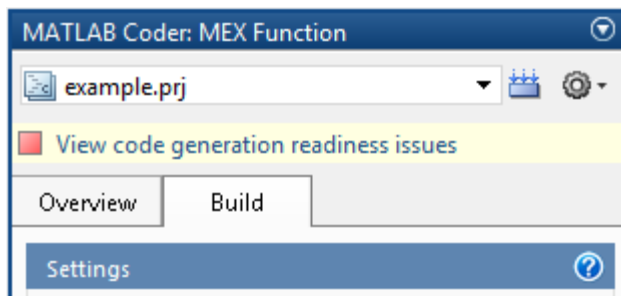
1 In the current folder browser, right-click the file that you want to check for code generation readiness.

2 From the context menu, select **Check Code Generation Readiness**.

The **Code Generation Readiness** tool opens for the selected file. It provides a code generation readiness score and lists issues that must be fixed prior to code generation.

Run the Code Generation Readiness Tool in a Project

- 1 After you have added entry-point files to the project, if MATLAB Coder detects code generation issues, it displays a link at the top of the project window.



- 2 Click the link to open the **Code Generation Readiness** tool.

The tool opens and provides a code generation readiness score and lists issues that must be fixed prior to code generation.

See Also

- “Code Generation Readiness Tool” on page 17-10

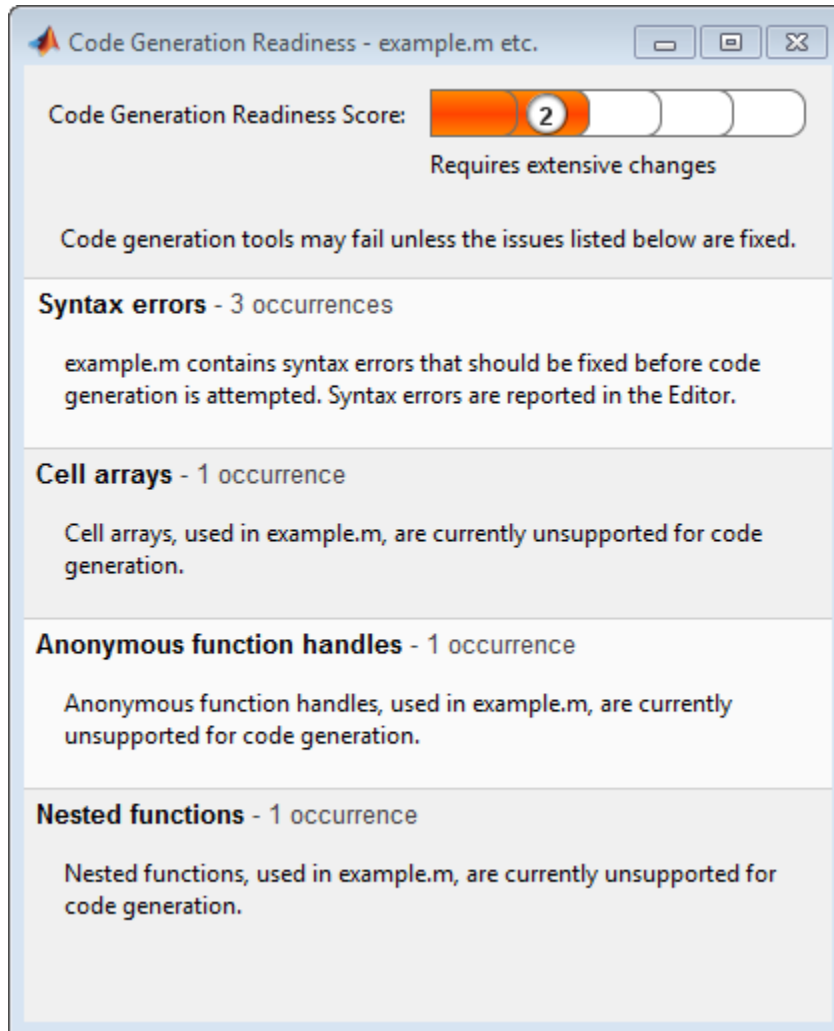
Code Generation Readiness Tool

In this section...
“What Information Does the Code Generation Readiness Tool Provide?” on page 17-10
“Summary Tab” on page 17-11
“Code Structure Tab” on page 17-12
“See Also” on page 17-15

What Information Does the Code Generation Readiness Tool Provide?

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code suitable for code generation. The tool might not detect all code generation issues. Under certain circumstances, it might report false errors. Because the tool might not detect all issues, or might report false errors, generate a MEX function to verify that your code is suitable for code generation before generating C code.

Summary Tab

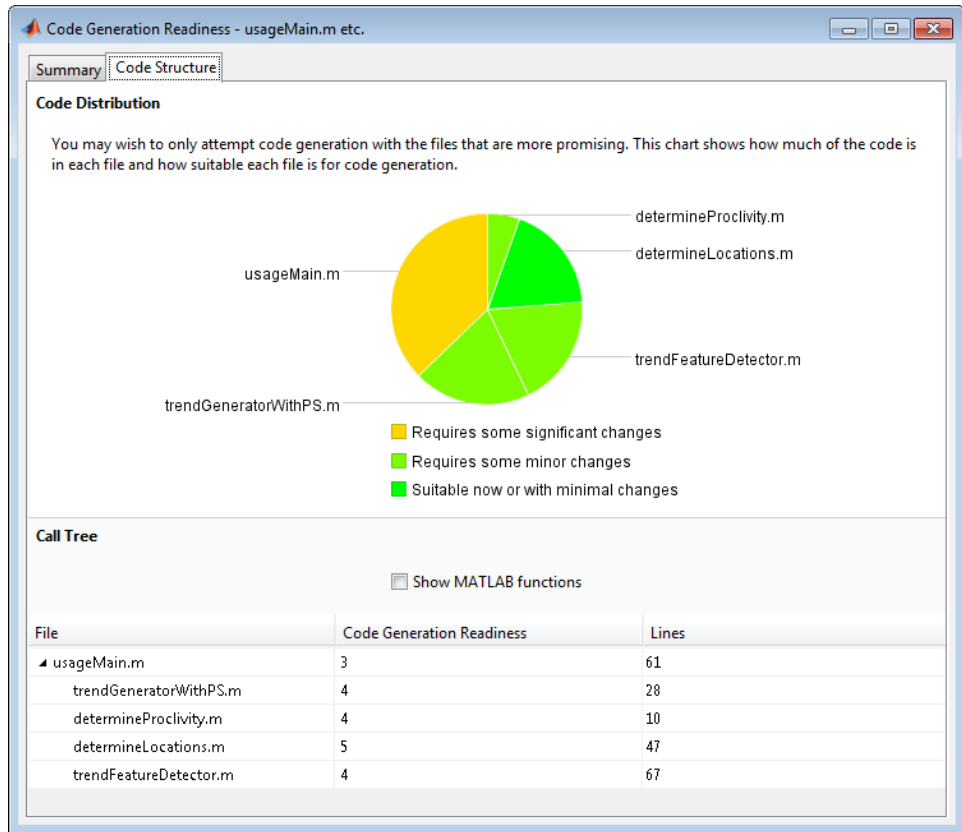


The **Summary** tab provides a **Code Generation Readiness Score** which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes.

On this tab, the tool also provides information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. Use the code analyzer to learn more about the issues and how to fix them.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features, such as recursion, cell arrays, nested functions, and function handles.
- Unsupported data types.

Code Structure Tab



If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab provides information about the relative size of each file and how suitable each file is for code generation.

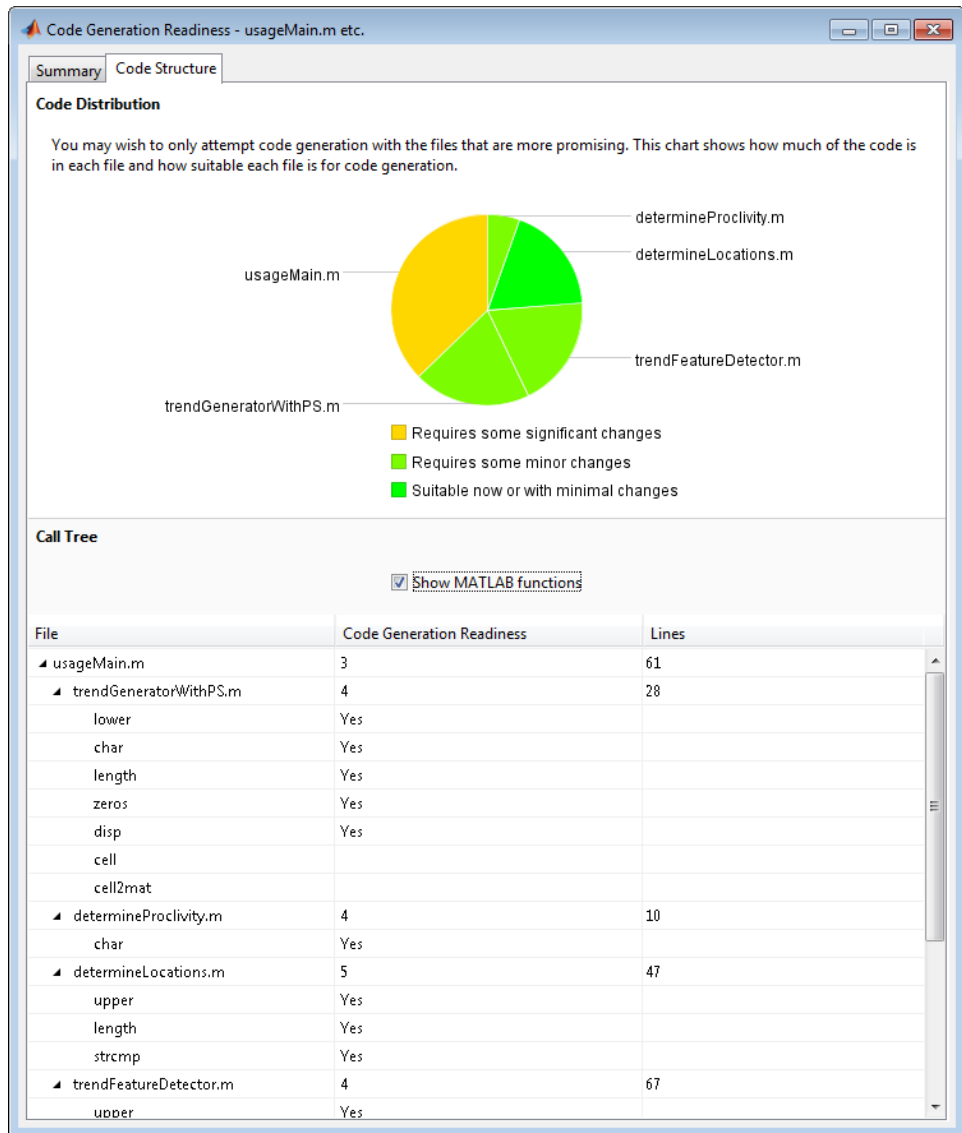
Code Distribution

The **Code Distribution** pane provides a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. This information is useful during the planning phase of a project for estimation and scheduling purposes. If the report indicates that there are multiple files not yet suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane provides information on the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions. If you select **Show MATLAB Functions**, the report also lists the MATLAB functions called by your function code. For each of these MATLAB functions, if the function is supported for code generation, the report sets **Code Generation Readiness** to Yes.



See Also

- “Check Code Using the Code Generation Readiness Tool” on page 17-8

Unable to Determine Code Generation Readiness

Sometimes the code generation readiness tool cannot determine whether the entry-point functions in your project are suitable for code generation. The most likely reason is that the tool is unable to find the entry-point files. Verify that your current working folder is set to the folder that contains your entry-point files. If it is not, either make this folder your current working folder or add the folder containing these files to the MATLAB path.

Generate MEX Functions Using the MATLAB Coder Project Interface

In this section...

“Project Workflow for Generating MEX Functions” on page 17-17

“Generate MEX Functions Using the Project Interface” on page 17-17

“Configure Project Settings” on page 17-22

“Build a MATLAB® Coder™ Project” on page 17-23

“See Also” on page 17-24

Project Workflow for Generating MEX Functions

Step	Action	Details
1	Set up your MATLAB Coder project.	“Creating a New Project” on page 16-3
2	Fix errors detected by the code analyzer.	“Fixing Errors Detected at Design Time” on page 17-4
3	Specify build configuration parameters.	“Configure Project Settings” on page 17-22
4	Build the project.	“Build a MATLAB® Coder™ Project” on page 17-23

Generate MEX Functions Using the Project Interface

In this example, you create a MATLAB function that adds two numbers, then create a MATLAB Coder project for this file. Using the project user interface, you specify types for the function input parameters, and then generate a MEX function for the MATLAB code.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

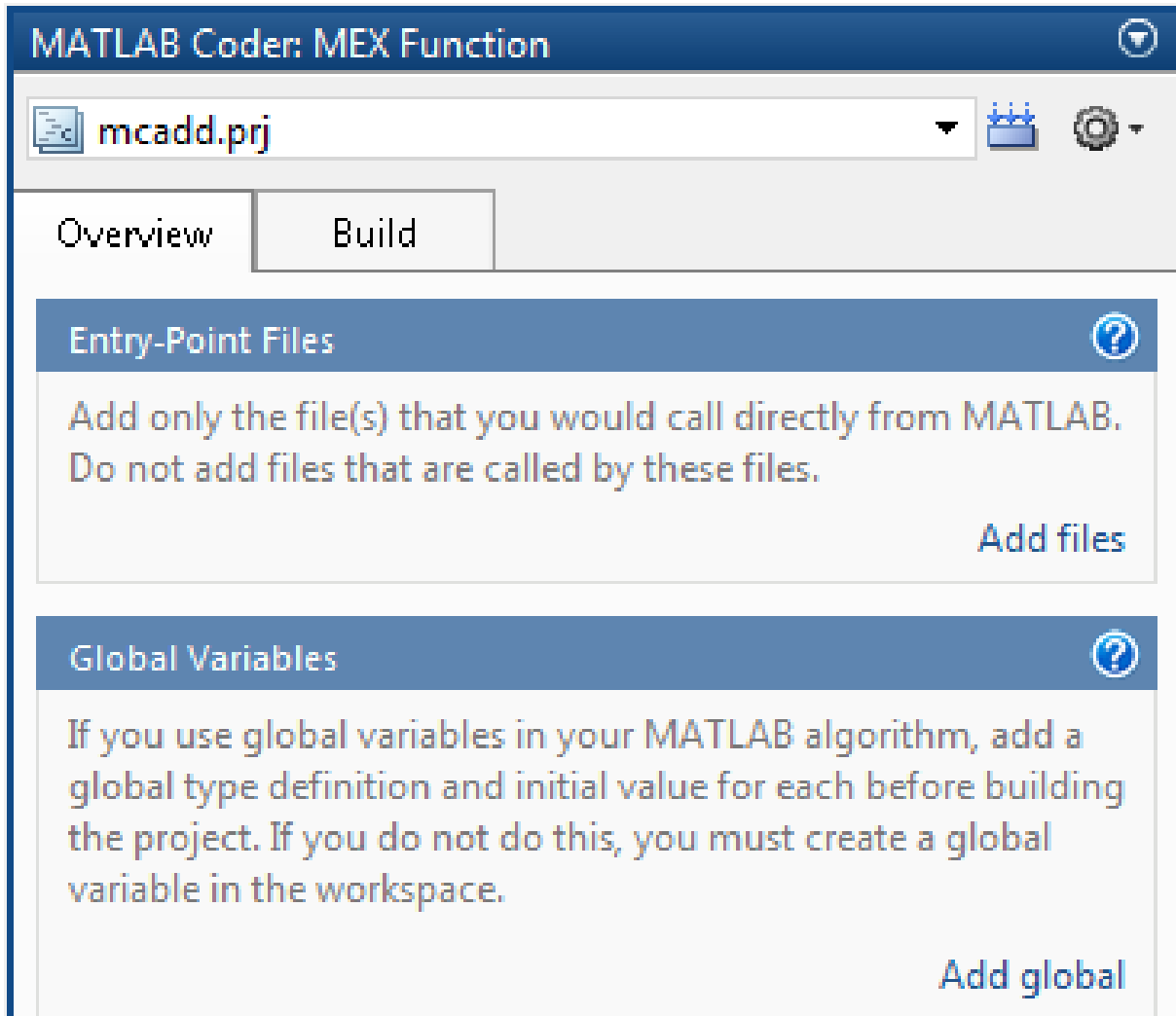
```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 In the same folder, set up a MATLAB Coder project.

- At the MATLAB command line, enter

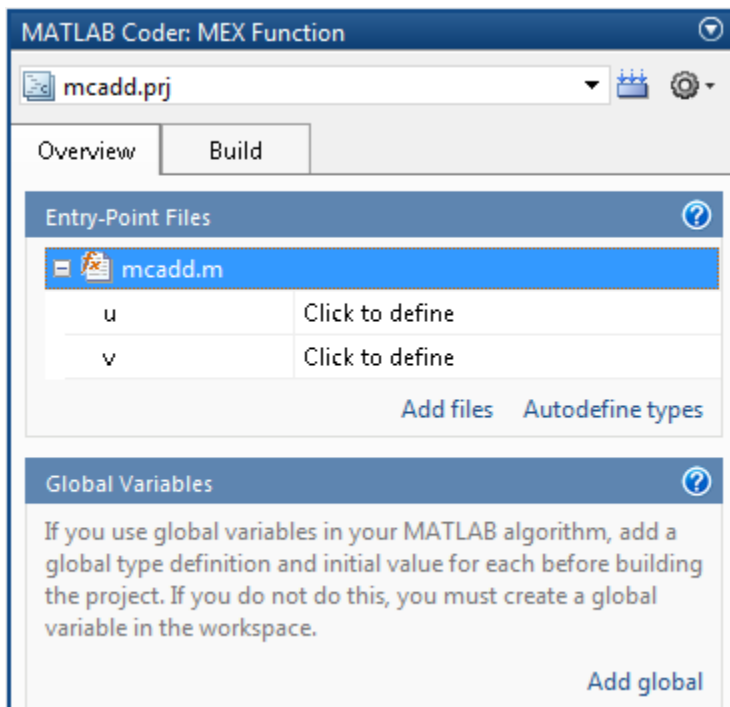
```
coder -new mcadd.prj
```

By default, the project opens in the MATLAB workspace on the right side.

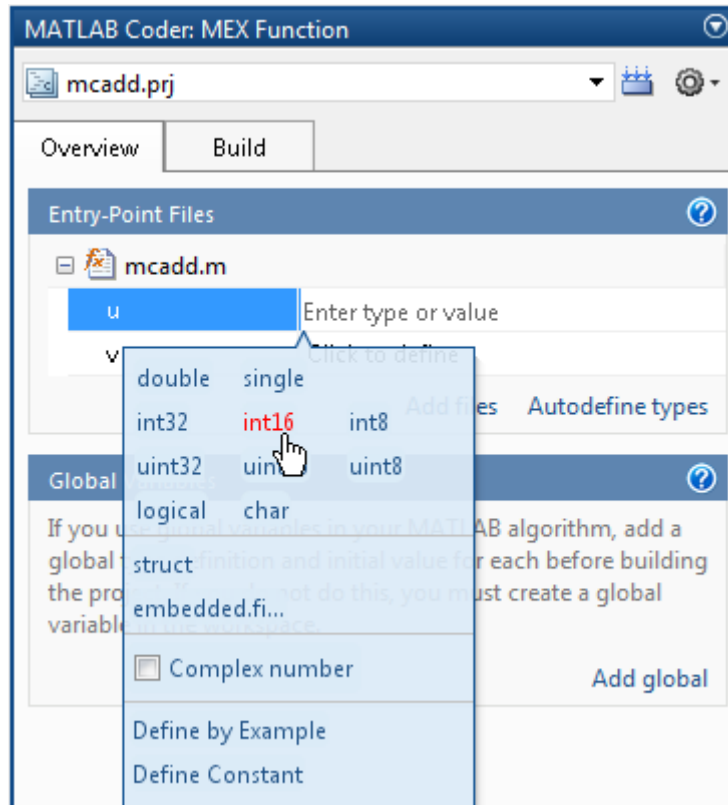


- b** On the project **Overview** tab, click the **Add files** link, browse to the file `mcadd.m`, and click **OK** to add the file to the project.

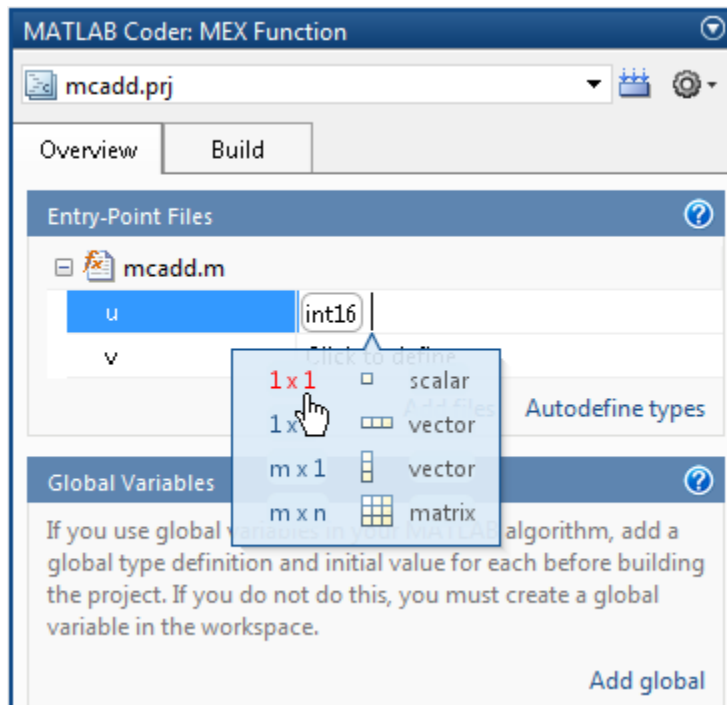
The file is displayed on the **Overview** tab, and both inputs are undefined.



- c** On the **Overview** tab, click the field to the right of the input parameter `u` and, from the list of input options, select `int16`.



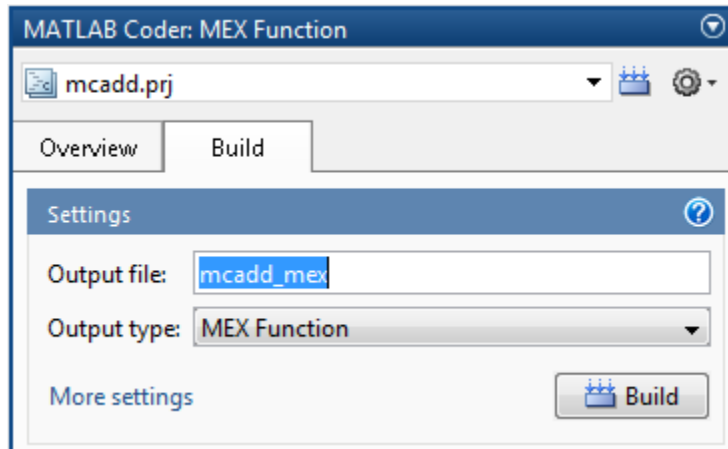
- d From the list of size options, select 1 x 1 to specify that the input is a scalar.



e Repeat the previous two steps to specify the input v .

3 In the MATLAB Coder project, click the **Build** tab.

By default, the **Output type** is MEX function and the **Output file name** is mcadd_mex.



- 4 On this tab, click the **Build** button to generate a MEX function using the default project settings.

MATLAB Coder builds the project and, by default, generates a MEX function, `mcadd_mex`, in the current folder. MATLAB Coder also generates other supporting files in a subfolder called `codegen/mexfcn/mcadd`. MATLAB Coder uses the name of the MATLAB function as the root name for the generated files and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 19-70.

You can now test your MEX function in MATLAB. For more information, see “Verify MEX Functions in a Project” on page 18-6.

Configure Project Settings

- 1 On the project **Build** tab, click the **More settings** link to view the project settings for the selected output type.

Note MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you change the output type from MEX Function or Instrumented MEX Function to C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

- 2 In the **Project Settings** dialog box, select the settings that you want to apply.

Tip To learn more about the configuration parameters on the current tab of the **Project Settings** dialog box, click the **Help** button.

See Also

- “How to Enable Code Generation Reports in the Project Settings Dialog Box” on page 19-179
- “In the Project Settings Dialog Box” on page 19-119
- “How to Disable Inlining Globally in the Project Settings Dialog Box” on page 19-130
- “Generate Traceable Code” on page 19-89
- “Disabling Run-Time Checks in the Project Settings Dialog Box” on page 22-19

Build a MATLAB Coder Project

On the project **Build** tab, click the **Build** button to build the project using the specified settings. While MATLAB Coder builds a project, it displays the build progress in the Build dialog box. When the build is complete, MATLAB Coder provides details in the **Build Results** pane.

Viewing Build Results

The **Build Results** pane provides information about the most recent build. If the code generation report is enabled or build errors occur, MATLAB Coder generates a report that provides detailed information about the most recent build and provides a link to the report.

To view the report, click the **View report** link. After a build completes, this report provides links to your MATLAB code and generated C/C++ files as well as compile-time type information for the variables in your MATLAB code. If build errors occur, it lists errors and warnings.

Saving Build Results

When MATLAB Coder builds a project, it displays the build progress and results in the Build dialog box. To save the build results, click the **Save to log file** link and specify the log file location.

See Also

- “Code Generation Reports” on page 19-176
- “Generate Code for Multiple Entry-Point Functions” on page 19-75
- “Generate Code for Global Data” on page 19-81

See Also

- “Generate Code for Multiple Entry-Point Functions” on page 19-75
- “Generate Code for Global Data” on page 19-81
- “Specify Output File Name” on page 16-40
- “Specify Output File Locations” on page 16-41

Generate MEX Functions at the Command Line

Command-line Workflow for Generating MEX Functions

Step	Action	Details
1	Install prerequisite products.	“Installing Prerequisite Products”
2	Set up your C/C++ compiler.	“Setting Up the C/C++ Compiler”
3	Set up your file infrastructure.	“Paths and File Infrastructure Setup” on page 19-69
4	Fix errors detected by the code analyzer.	“Fixing Errors Detected at Design Time” on page 17-4
5	Specify build configuration parameters.	“Specify Build Configuration Parameters” on page 19-28
6	Specify properties of primary function inputs.	“Primary Function Input Specification” on page 19-38
7	Generate the MEX function using codegen with suitable command-line options.	“Generating MEX Functions at the Command Line Using codegen” on page 17-26

Generate MEX Functions at the Command Line

In this example, you use the `codegen` function to generate a MEX function from a MATLAB file that adds two inputs. You use the `codegen -args` option to specify that both inputs are `int16`.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 Generate a platform-specific MEX function in the current folder. At the command line, specify that the two input parameters are `int16` using the `-args` option. By default, if you do not use the `-args` option, `codegen` treats inputs as real, scalar doubles.

```
codegen mcadd -args {int16(0), int16(0)}
```

`codegen` generates a MEX function, `mcadd_mex`, in the current folder. `codegen` also generates other supporting files in a subfolder called `codegen/mexfcn/mcadd`. `codegen` uses the name of the MATLAB function as the root name for the generated files and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 19-70.

Generating MEX Functions at the Command Line Using `codegen`

You generate a MEX function at the command line using the `codegen` function.

The basic command is:

```
codegen fcn
```

By default, `codegen` generates a MEX function in the current folder as described in “Generate MEX Functions at the Command Line” on page 17-25.

You can modify this default behavior by specifying one or more compiler options with `codegen`, separated by spaces on the command line. For more information, see `codegen`.

See Also

- “Primary Function Input Specification” on page 19-38
- “MEX Function Generation at the Command Line”
- “Generate Code for Multiple Entry-Point Functions” on page 19-75
- “Generate Code for Global Data” on page 19-81

Fix Errors Detected at Code Generation Time

When the code generation software detects errors or warnings, it automatically generates an error report. The error report describes the issues and provides links to the MATLAB code with errors.

To fix the errors, modify your MATLAB code to use only those MATLAB features that are supported for code generation. For more information, see “MATLAB Algorithm Design Basics”. Choose a debugging strategy for detecting and correcting code generation errors in your MATLAB code. For more information, see “Debugging Strategies” on page 17-31.

When code generation is complete, the software generates a MEX function that you can use to test your implementation in MATLAB.

If your MATLAB code calls functions on the MATLAB path, unless the code generation software determines that these functions should be extrinsic or you declare them to be extrinsic, it attempts to compile these functions. See “Resolution of Function Calls in MATLAB Generated Code” on page 13-2. To get detailed diagnostics, add the `%#codegen` directive to each external function that you want `codegen` to compile.

See Also

- “Code Generation Reports” on page 19-176
- “Why Test MEX Functions in MATLAB?” on page 18-4
- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Debugging Strategies” on page 17-31
- “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12

Design Considerations When Writing MATLAB Code for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

You can choose whether the generated code uses static or dynamic memory allocation.

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get best speed, but with higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- Speed

Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

To improve the speed of the generated code:

- Choose a suitable C/C++ compiler. The default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms is not a good compiler for performance.

- Consider disabling run-time checks.

By default, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower MEX function execution. Disabling run-time checks usually results in streamlined generated code and faster MEX function execution. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

See Also

- “MATLAB Algorithm Design Basics”
- “Data Definition”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data” on page 7-4
- “Control Dynamic Memory Allocation” on page 19-99
- “Control Run-Time Checks” on page 22-18

Running MEX Functions

When you call a MEX function, pass it the same inputs you use for the original MATLAB algorithm. Do not pass `coder.Constant` or any of the `coder.Type` classes to a MEX function; these classes are for use with the `codegen` function.

To run a MEX function generated by MATLAB Coder, you must have licenses for all the toolboxes that the MEX function requires. For example, if you generate a MEX function from a MATLAB algorithm that uses a Computer Vision System Toolbox function or System object, to run the MEX function, you must have a Computer Vision System Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

Debugging MEX Functions

You cannot use the `disp` and `save` functions during debugging to inspect the contents of your MEX function variables. Because these functions are not supported for code generation, you must declare them as extrinsic functions. For extrinsic functions, when running the MEX function, MATLAB Coder calls out to MATLAB to run `disp` and `save`, so they save and display the data found in the base workspace, not the MEX-function workspace.

Debugging Strategies

Before you perform code verification, choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other's functions. The following table describes two general strategies, each of which has advantages and disadvantages.

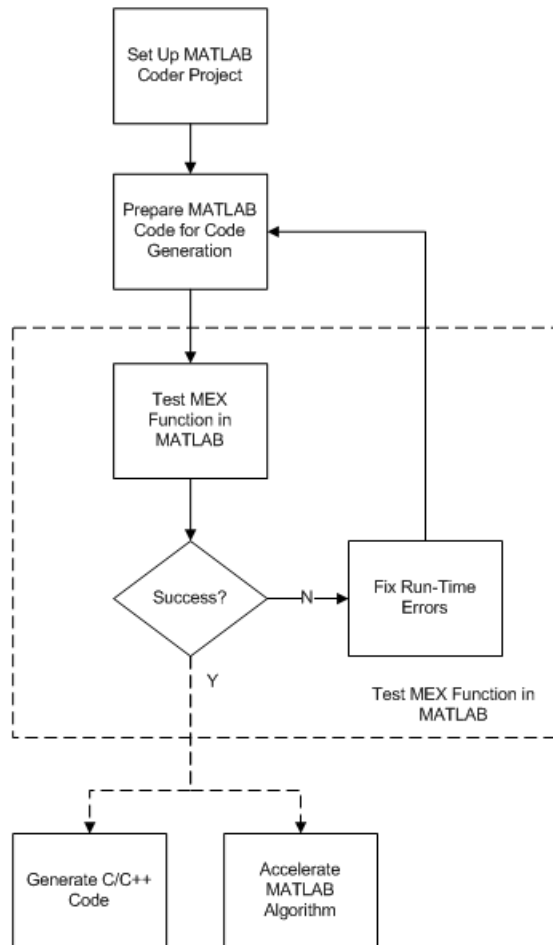
Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	<ol style="list-style-type: none">1 Verify that your lowest-level (leaf) functions are compliant.2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function.	<ul style="list-style-type: none">• Efficient• Unlikely to cause errors• Easy to isolate code generation syntax violations	Requires application tests that work from the bottom up

Debugging Strategy	What to Do	Pros	Cons
Top-down verification	<ol style="list-style-type: none"> 1 Declare functions called by the top-level function to be extrinsic so that MATLAB Coder does not compile them. See “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12. 2 Verify that your top-level function is compliant. 3 Work your way down the function hierarchy incrementally by removing extrinsic declarations one by one to compile and verify each function, ending with the leaf functions. 	You retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: <ul style="list-style-type: none"> • Extrinsic declarations • Additional assignment statements as required to convert opaque values returned by extrinsic functions to nonopaque values (see “Working with mxArray” on page 13-17).

Testing MEX Functions in MATLAB

- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “Why Test MEX Functions in MATLAB?” on page 18-4
- “Running MEX Functions” on page 18-5
- “Verify MEX Functions in a Project” on page 18-6
- “Verify MEX Functions at the Command Line” on page 18-8
- “Debug Run-Time Errors” on page 18-9

Workflow for Testing MEX Functions in MATLAB



See Also

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2

- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Why Test MEX Functions in MATLAB?” on page 18-4
- “Debug Run-Time Errors” on page 18-9
- “C/C++ Code Generation” on page 19-5
- “Accelerate MATLAB Algorithms” on page 22-11

Why Test MEX Functions in MATLAB?

Before generating C/C++ code for your MATLAB code, it is a best practice to test the MEX function to verify that it provides the same functionality as the original MATLAB code. To do this testing, run the MEX function using the same inputs as you used to run the original MATLAB code and compare the results. For more information about how to test a MEX function in a project, see “Verify MEX Functions in a Project” on page 18-6. For more information on how to test a MEX function at the command line, see “Verify MEX Functions at the Command Line” on page 18-8.

In addition, running the MEX function in MATLAB before generating code enables you to detect and fix run-time errors that are much harder to diagnose in the generated code. If you encounter run-time errors in your MATLAB functions, fix them before generating code. For more information, see “Debug Run-Time Errors” on page 18-9.

When you run your MEX function in MATLAB, by default, the following run-time checks execute :

- Memory integrity checks. These checks perform array bounds checking, dimension checking, and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution and provides a diagnostic message.
- Responsiveness checks in code generated for MATLAB functions. These checks enable periodic checks for **Ctrl+C** breaks in code generated for MATLAB functions, allowing you to terminate execution with **Ctrl+C**.

For more information, see “Control Run-Time Checks” on page 22-18.

Running MEX Functions

When you call a MEX function, pass it the same inputs you use for the original MATLAB algorithm. Do not pass `coder.Constant` or any of the `coder.Type` classes to a MEX function; these classes are for use with the `codegen` function.

To run a MEX function generated by MATLAB Coder, you must have licenses for all the toolboxes that the MEX function requires. For example, if you generate a MEX function from a MATLAB algorithm that uses a Computer Vision System Toolbox function or System object, to run the MEX function, you must have a Computer Vision System Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

Debugging MEX Functions

You cannot use the `disp` and `save` functions during debugging to inspect the contents of your MEX function variables. Because these functions are not supported for code generation, you must declare them as extrinsic functions. For extrinsic functions, when running the MEX function, MATLAB Coder calls out to MATLAB to run `disp` and `save`, so they save and display the data found in the base workspace, not the MEX-function workspace.

Verify MEX Functions in a Project

In this section...

“Using Test Files That Call Only MATLAB Functions” on page 18-6


“Using Test Files That Call MEX Functions” on page 18-7

Using Test Files That Call Only MATLAB Functions

If you have a test file that calls only your original entry-point MATLAB function, use the following procedure. A test file can be either a MATLAB function or a script. To use this procedure, you should verify that it calls at least one entry-point function. The generated MEX function must be in the same folder as the entry-point functions.

Selecting the **Redirect entry-point calls to MEX function** option directs MATLAB Coder software to replace calls to the MATLAB function with calls to the generated MEX function. This capability allows you to compare the behavior of the MEX function with that of the original function.

If your test file calls the generated MEX function, do not follow this procedure. Instead, follow the procedure in “Using Test Files That Call MEX Functions” on page 18-7.

- 1 On the project **Build** tab **Verification** panel, click the  button to add a test file. Alternatively, if you have already added test files to the project, select one from the list.
- 2 Run the test file calling the original MATLAB algorithm.
 - a Clear **Redirect entry-point calls to MEX function**.
 - b Click the **Run** button.

The test file runs and calls your original MATLAB algorithm.

- 3 Verify that the test results are as expected.
- 4 Run the test file calling the MEX function instead of the original MATLAB algorithm.

- a Select **Redirect entry-point calls to MEX function**.
- b Click the **Run** button.


The project builds the MEX function. The test file runs and automatically replaces calls to your original MATLAB algorithm with calls to the generated MEX function.

- 5 Compare the results of the two runs to verify that the MEX function provides the same functionality as the original MATLAB algorithm.

Using Test Files That Call MEX Functions

If you have a test file that calls the generated MEX function, use the following procedure. If your test file calls both the original MATLAB function and the generated MEX function, you can also use this procedure.

A test file can be either a MATLAB function or a script. To use this procedure, you should verify that it calls at least one MEX function. The MEX function must be in the same folder as the entry-point functions.

- 1 On the project **Build** tab **Verification** panel, click the  button to add a test file. Alternatively, if you have already added test files to the project, select one from the list.

- 2 Run the test file.

- a Clear **Redirect entry-point calls to MEX function**.

Because the test file already calls the MEX function, you do not want MATLAB Coder to redirect entry-point function calls.

- b Click the **Run** button.

The project builds the MEX function. The test file runs and calls the generated MEX function. If applicable, it also calls the original MATLAB algorithm.

- 3 Use the results of this run to verify that the MEX function provides the same functionality as the original MATLAB algorithm.

Verify MEX Functions at the Command Line

If you have a test file that calls your original MATLAB function, use `coder.runTest` to verify the MEX function at the command line. `coder.runTest` runs the test file replacing calls to the original MATLAB function with calls to the generated MEX function. If errors occur during the run with `coder.runTest`, call stack information is available for debugging purposes. For more information, see the `coder.runTest` function reference information and “Verifying the MEX Function” in the MATLAB Coder “C Code Generation at the Command Line” tutorial.

Debug Run-Time Errors

In this section...
“Viewing Errors in the Run-Time Stack” on page 18-9
“Handling Run-Time Errors” on page 18-11

If you encounter run-time errors in your MATLAB functions, the run-time stack appears automatically in the MATLAB command window. Use the error message and stack information to learn more about the source of the error and then either fix the issue or add error-handling code. For more information, see “Viewing Errors in the Run-Time Stack” on page 18-9 “Handling Run-Time Errors” on page 18-11.

Viewing Errors in the Run-Time Stack

About the Run-Time Stack

The run-time stack is enabled by default for MEX code generation from MATLAB. Use the error message and the following stack information to learn more about the source of the error:

- The name of the function that generated the error
- The line number of the attempted operation
- The sequence of function calls that led up to the execution of the function and the line at which each of these function calls occurred

Example Run-Time Stack Trace. This example shows the run-time stack trace for MEX function `mlstack_mex`:

```
mlstack_mex(-1)

Index exceeds matrix dimensions. Index
value -1 exceeds valid range [1-4] of
array x.

Error in mlstack>mayfail (line 31)
y = x(u);

Error in mlstack>subfcn1 (line 5)
switch (mayfail(u))

Error in mlstack (line 2)
y = subfcn1(u);
```

The stack trace provides the following information:

- The type of error.

```
??? Index exceeds matrix dimensions.
Index value -1 exceeds valid range [1-4] of array x.
```
- Where the error occurred.

```
Error in ==>mlstack>mayfail at 31
y = x(u);
```
- The function call sequence prior to the failure.

```
Error in ==> mlstack>subfcn1 at 5
switch (mayfail(u))

Error in ==> mlstack at 2
y = subfcn1(u);
```

When to Use the Run-Time Stack

The run-time stack is useful during debugging to help you find the source of run-time errors. However, when the stack is enabled, the generated code

contains instructions for maintaining the run-time stack, which might slow the run time. Consider disabling the run-time stack for faster run time.

How to Disable the Run-Time Stack. You can disable the run-time stack by disabling the memory integrity checks as described in “How to Disable Run-Time Checks” on page 22-19.

Caution Before disabling the memory integrity checks, you should verify that all array bounds and dimension checking is unnecessary.

Handling Run-Time Errors

The code generation software propagates error ID's. If you throw an error or warning in your MATLAB code, use the `try-catch` statement in your test bench code to examine the error information and attempt to recover, or clean up and abort. For example, for the function in “Example Run-Time Stack Trace” on page 18-10, create a test script containing:

```
try
    mlstack_mex(u)
catch
    % Add your error handling code here
end
```

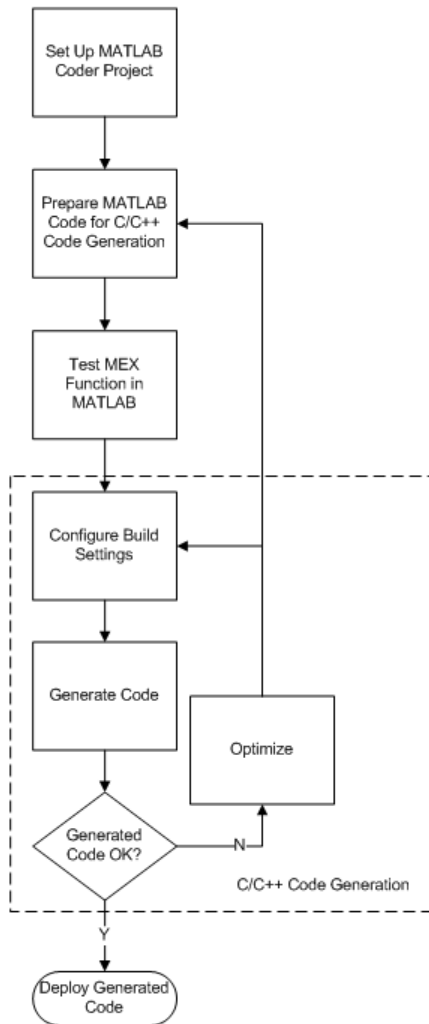
For more information, see “The try/catch Statement”.

Generating C/C++ Code from MATLAB Code

- “Code Generation Workflow” on page 19-3
- “C/C++ Code Generation” on page 19-5
- “Generating C/C++ Static Libraries from MATLAB Code” on page 19-7
- “Generating C/C++ Dynamically Linked Libraries from MATLAB Code” on page 19-11
- “Generating Standalone C/C++ Executables from MATLAB Code” on page 19-15
- “Build Setting Configuration” on page 19-21
- “Share Build Configuration Settings” on page 19-35
- “Primary Function Input Specification” on page 19-38
- “Define Input Properties Programmatically in the MATLAB File” on page 19-50
- “Speed Up Compilation” on page 19-61
- “Code Optimization” on page 19-63
- “Paths and File Infrastructure Setup” on page 19-69
- “Generate Code for Multiple Entry-Point Functions” on page 19-75
- “Generate Code for Global Data” on page 19-81
- “Generation of Traceable Code” on page 19-88
- “Generate Code for Enumerated Types” on page 19-97
- “Generate Code for Variable-Size Data” on page 19-98

- “Code Generation for MATLAB Classes” on page 19-118
- “How MATLAB® Coder™ Partitions Generated Code” on page 19-119
- “Customize the Post-Code-Generation Build Process” on page 19-133
- “Code Generation Reports” on page 19-176
- “Troubleshooting” on page 19-195
- “Package Code For Use in Another Development Environment” on page 19-196

Code Generation Workflow



See Also

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “Build Setting Configuration” on page 19-21
- “C/C++ Code Generation” on page 19-5
- “Code Optimization” on page 19-63

C/C++ Code Generation

Using MATLAB Coder, you can generate standalone C/C++ static and dynamic libraries and C/C++ executables. If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes. By default, if MATLAB Coder does not detect errors, it generates a platform-specific MEX function in the current folder.

To learn how to generate...	See...
C/C++ static libraries from your MATLAB code	“Generating C/C++ Static Libraries from MATLAB Code” on page 19-7
C/C++ dynamic libraries from your MATLAB code	“Generating C/C++ Dynamically Linked Libraries from MATLAB Code” on page 19-11
C/C++ executables from your MATLAB code	“Generating Standalone C/C++ Executables from MATLAB Code” on page 19-15
MEX functions from your MATLAB code	“Generate MEX Functions Using the MATLAB® Coder™ Project Interface” on page 17-17

If errors occur, MATLAB Coder does not generate code, but produces an error report and provides a link to this report. For more information, see “Code Generation Reports” on page 19-176.

Specify Custom Files to Build

In addition to your MATLAB file, you can specify the following types of custom *files* to include in the build for standalone C/C++ code generation.

File Extension	Description
.c	Custom C file
.cpp	Custom C++ file
.h	Custom header file

File Extension	Description
.o , .obj	Custom object file
.a , .lib, .so	Library
.tmf	Template makefile for custom MATLAB Coder builds

Generating C/C++ Static Libraries from MATLAB Code

In this section...

“Generate a C Static Library Using the Project Interface” on page 19-7

“Generate a C Static Library at the Command Line” on page 19-10

Generate a C Static Library Using the Project Interface

This example shows how to generate a C static library from MATLAB code using a MATLAB Coder project.

In this example, you create a MATLAB function that adds two numbers. You then create a MATLAB Coder project. Use the project user interface to generate a C static library for the MATLAB code.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 In the same folder, set up a MATLAB Coder project.

- a At the MATLAB command line, enter:

```
coder -new mcadd.prj
```

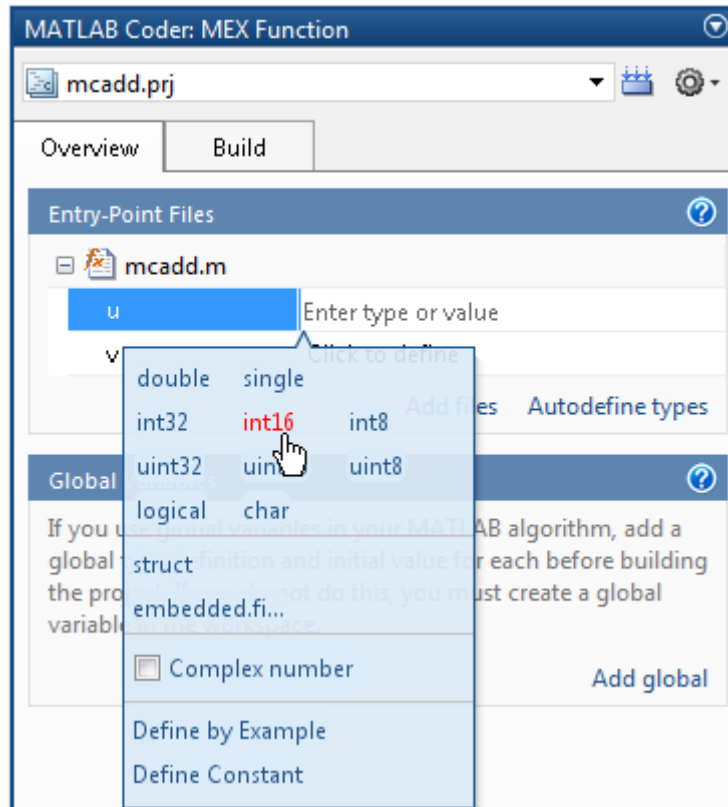
By default, the project opens in the MATLAB workspace on the right side.

- b On the project **Overview** tab, click the **Add files** link. Browse to the file `mcadd.m`. Click **OK** to add the file to the project.

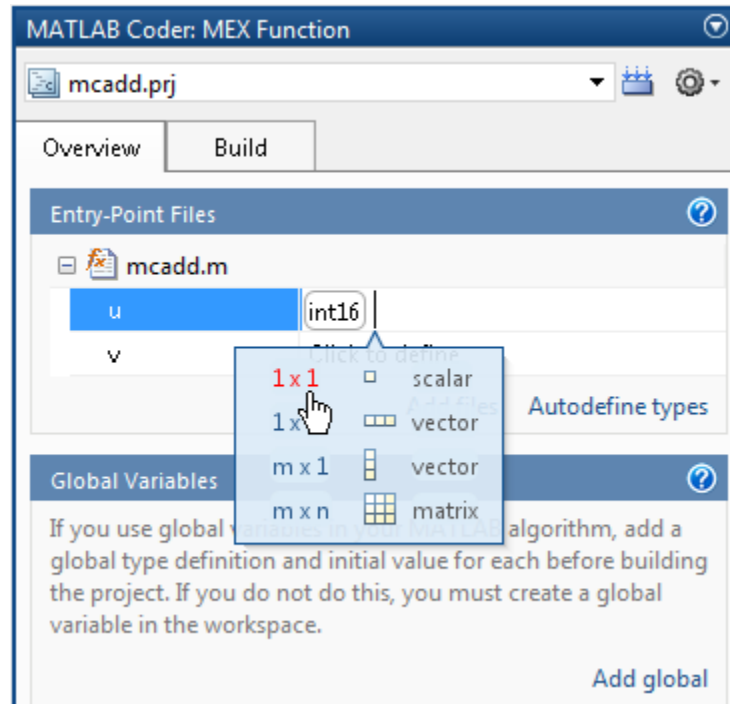
The file is displayed on the **Overview** tab. Both inputs are undefined.

- 3 Define the type of input `u`.

- a On the **Overview** tab, click the field to the right of the input parameter `u` and, from the list of input options, select `int16`.



- b** From the list of size options, select 1 x 1 to specify that the input is a scalar.



- 4 Repeat the previous step for input *v*.
- 5 In the MATLAB Coder project, click the **Build** tab.
- 6 On this tab, set the **Output type** to **C/C++ Static library**.
The default output file name is `mcadd`.
- 7 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default folder, `codegen/lib/mcadd`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Generate a C Static Library at the Command Line

This example shows how to generate a C static library from MATLAB code at the command line using the `codegen` function.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 Using the `config:lib` option, generate C library files. Using the `-args` option, specify that the first input is a 1-by-4 vector of unsigned 16-bit integers and that the second input is a double-precision scalar.

```
codegen -config:lib mcadd -args {zeros(1,4,'uint16'),0}
```

MATLAB Coder generates a C static library with the default name, `mcadd`, and supporting files in the default folder, `codegen/lib/mcadd`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Generating C/C++ Dynamically Linked Libraries from MATLAB Code

In this section...

“Dynamic Libraries Generated by MATLAB® Coder™” on page 19-11

“Generate a C Dynamically Linked Library (DLL) Using the Project Interface” on page 19-11

“Generate a C Dynamic Library at the Command Line” on page 19-13

Dynamic Libraries Generated by MATLAB Coder

By default, when MATLAB Coder generates a dynamic library (DLL):

- The DLL is suitable for the platform that you are working on.
- The DLL uses the release version of the C run-time library.
- The DLL linkage conforms to the target language, by default, C. If you set the target language to C++, the linkage conforms to C++.
- When the target language is C, the generated header files explicitly declare the exported functions to be `extern "C"` to simplify integration of the DLL into C++ applications.

If you generate a DLL that uses dynamically allocated variable-size data, MATLAB Coder automatically provides exported utility functions to interact with this data in the generated code. For more information, see “Utility Functions for Creating `emxArray` Data Structures” on page 7-21.

Generate a C Dynamically Linked Library (DLL) Using the Project Interface

This example shows how to generate a C DLL from MATLAB code using a MATLAB Coder project.

In this example, you create a MATLAB function that generates a random scalar value. You then create a MATLAB Coder project. Use the project user interface to generate a C dynamic library for the MATLAB code.

- 1 Write two MATLAB functions, `ep1` takes one input, a single scalar, and `ep2` takes two inputs, both double scalars. In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

In the same folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

- 2 In the same folder as the `ep1` and `ep2` files, set up a MATLAB Coder project. At the MATLAB command line, enter:

```
coder -new ep.prj
```

By default, the project opens in the MATLAB workspace on the right side.

- 3 On the project **Overview** tab, click the **Add files** link and browse to the file `ep1.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. MATLAB Coder indicates that input `u` is undefined.

- 4 Define the type of input `u`.

- a On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select **single**.

- b From the list of size options, select `1 x 1` to specify that `u` is a scalar.

- 5 On the project **Overview** tab, click the **Add files** link and browse to the file `ep2.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. MATLAB Coder indicates that inputs `u` and `v` are undefined.

- 6 Define the type of input `u`.

- a On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select **double**.

- b From the list of size options, select `1 x 1` to specify that `u` is a scalar.

- 7 Repeat the previous step for input *v*.
- 8 In the MATLAB Coder project, click the **Build** tab.
- 9 On the **Build** tab, set the **Output type** to C/C++ Dynamic Library.
- 10 On the **Build** tab, click the **Build** button to generate a library using these project settings.

On Microsoft® Windows systems, MATLAB Coder generates a C dynamic library, `ep1.dll`, and supporting files, in the default folder, `codegen/dll/ep1`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library. On Linux® and Macintosh systems, it generates a shared object (`.so`) file. The DLL linkage conforms to the target language, in this example, C. If you set the target language to C++, the linkage conforms to C++.

Generate a C Dynamic Library at the Command Line

This example shows how to generate a C dynamic library from MATLAB code at the command line using the `codegen` function.

- 1 Write two MATLAB functions, `ep1` takes one input, a single scalar, and `ep2` takes two inputs, both double scalars. In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

In the same folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

- 2 Generate the C dynamic library.

```
codegen -config:dll ep1 -args single(0) ep2 -args {0,0}
```

On Microsoft Windows systems, `codegen` generates a C dynamic library, `ep1.dll`, and supporting files, in the default folder, `codegen/dll/ep1`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library. On Linux and Macintosh systems,

it generates a shared object (.so) file. The DLL linkage conforms to the target language, in this example, C. If you set the target language to C++, the linkage conforms to C++.

Note The default target language is C. To change the target language to C++, see “Specify a Language for Code Generation” on page 19-24.

Generating Standalone C/C++ Executables from MATLAB Code

In this section...

“Generate a C Executable Using the Project Interface” on page 19-15

“Generate a C Executable at the Command Line” on page 19-17

“Specifying main Functions for C/C++ Executables” on page 19-19

“Specify main Functions” on page 19-19

Generate a C Executable Using the Project Interface

In this example, you create a MATLAB function that generates a random scalar value and a main C function that calls this MATLAB function. You then create a MATLAB Coder project. Use the project user interface to specify types for the function input parameters, specify the main function, and generate a C executable for the MATLAB code.

- 1 Write a MATLAB function, `coderand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1):

```
function r = coderand() %#codegen
r = rand();
```

- 2 Write a main C function, `c:\myfiles\main.c`, that calls `coderand`. For example:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_initialize.h"
#include "coderand_terminate.h"

int main()
{
    coderand_initialize();
```

```
    printf("coderand=%g\n", coderand());  
  
    coderand_terminate();  
  
    return 0;  
}
```

Note In this example, because the default file partitioning method is to generate one file for each MATLAB file, you include `coderand_initialize.h` and `coderand_terminate.h`. If your file partitioning method is set to generate one file for all functions, do **not** include `coderand_initialize.h` and `coderand_terminate.h`.

3 In the same folder as the `coderand` file, set up a MATLAB Coder project.

a At the MATLAB command line, enter:

```
coder -new coderand.prj
```

By default, the project opens in the MATLAB workspace on the right side.

b On the project **Overview** tab, click the **Add files** link and browse to the file `coderand.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. MATLAB Coder indicates that the `coderand` function has no inputs.

4 In the MATLAB Coder project, click the **Build** tab.

a Set the **Output type** to **C/C++ Executable**.

b Set the output file name to `coderand_exe`.

5 On the project **Build** tab, click the **More settings** link.

6 On the Project Settings dialog box **Custom Code** tab, under **Additional files and directories to be built**, set:

a **Source files** to `main.c`, which is the name of the C/C++ source file that contains the `main` function.

- b Include directories** to the location of `main.c`: `c:\myfiles`.
- c** Close the dialog box.

Note When you are building an executable, you must specify the main function. For more information, see “Specifying main Functions for C/C++ Executables” on page 19-19.

- 7** On the **Build** tab, click the **Build** button to generate a library using the default project settings.

MATLAB Coder compiles and links the main function with the C code that it generates for the project and, in the current folder, generates an executable, `coderand.exe`. It generates supporting files in the default folder, `codegen/exe/coderand`. MATLAB Coder generates the minimal set of `#include` statements for header files required by the selected code replacement library.

See Also

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “Build Setting Configuration” on page 19-21
- “C/C++ Code Generation” on page 19-5
- “Code Optimization” on page 19-63

Generate a C Executable at the Command Line

In this example, you create a MATLAB function that generates a random scalar value and a main C function that calls this MATLAB function. You then specify types for the function input parameters, specify the main function, and generate a C executable for the MATLAB code.

- 1** Write a MATLAB function, `coderand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1):

```
function r = coderand() %#codegen
r = rand();
```

- 2** Write a main C function, `c:\myfiles\main.c`, that calls `coderand`. For example:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_initialize.h"
#include "coderand_terminate.h"

int main()
{
    coderand_initialize();

    printf("coderand=%g\n", coderand());

    coderand_terminate();

    return 0;
}
```

Note In this example, because the default file partitioning method is to generate one file for each MATLAB file, you include `coderand_initialize.h` and `coderand_terminate.h`. If your file partitioning method is set to generate one file for all functions, do **not** include `coderand_initialize.h` and `coderand_terminate.h`.

- 3** Configure your code generation parameters to include the main C function and then generate the C executable:

```
cfg = coder.config('exe');
cfg.CustomSource = 'main.c';
cfg.CustomInclude = 'c:\myfiles';
codegen -config cfg coderand
```

codegen generates a C executable, `coderand.exe`, in the current folder. It generates supporting files in the default folder, `codegen/exe/coderand`. codegen generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Specifying main Functions for C/C++ Executables

When you generate an executable, you must provide a `main` function. If you are generating a C executable, provide a C file, `main.c`. If you are generating a C++ executable, provide a C++ file, `main.cpp`. Verify that the folder containing the main function has only one main file. Otherwise, `main.c` takes precedence over `main.cpp`, which causes an error when generating C++ code. You can specify the main file from the project settings dialog box, the command line, or the Code Generation dialog box.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder automatically generates an initialize function and a terminate function.

- If your file partitioning method is set to generate one file for each MATLAB file, you must include the initialize and terminate header functions in `main.c`. Otherwise, do not include them in `main.c`.
- You must call these functions along with the C/C++ function. For more information, see “Calling Initialize and Terminate Functions” on page 21-7.

Specify main Functions

Specifying main Functions in the Project Settings Dialog Box

- 1 On the project **Build** tab, click the **More settings** link to open the **Project Settings** dialog box.
- 2 On the **Custom Code** tab, set:
 - a **Additional source files** to the name of the C/C++ source file that contains the `main` function. For example, `main.c`. For more information, see “Specifying main Functions for C/C++ Executables” on page 19-19.
 - b **Additional include directories** to the location of `main.c`. For example, `c:\myfiles`.

Specifying main Functions at the Command Line

Set the `CustomSource` and `CustomInclude` properties of the code generation configuration object (see “Working with Configuration Objects” on page 19-30). The `CustomInclude` property indicates the location of C/C++ files specified by `CustomSource`.

- 1** Create a configuration object for an executable:

```
cfg = coder.config('exe');
```

- 2** Set the `CustomSource` property to the name of the C/C++ source file that contains the main function. (For more information, see “Specifying main Functions for C/C++ Executables” on page 19-19.) For example:

```
cfg.CustomSource = 'main.c';
```

- 3** Set the `CustomInclude` property to the location of `main.c`. For example:

```
cfg.CustomInclude = 'c:\myfiles';
```

- 4** Generate the C/C++ executable using the command line options. For example, if `myFunction` takes one input parameter of type `double`:

```
codegen -config cfg myFunction -args {0}
```

MATLAB Coder compiles and links the main function with the C/C++ code that it generates from `myFunction.m`.

Build Setting Configuration

In this section...

“Specify Output Type” on page 19-21

“Specify a Language for Code Generation” on page 19-24

“Specify Output File Name” on page 19-25

“Specify Output File Locations” on page 19-26

“Parameter Specification Methods” on page 19-27

“Specify Build Configuration Parameters” on page 19-28

Specify Output Type

Output Types

MATLAB Coder can generate code for the following output types:

- MEX function
- Instrumented MEX function
- Standalone C/C++ code and compile it to a static library
- Standalone C/C++ code and compile it to a dynamically-linked library
- Standalone C/C++ code and compile it to an executable

Note When you generate an executable, you must provide a C/C++ file that contains the `main` function, as described in “Specifying main Functions for C/C++ Executables” on page 19-19.

Location of Generated Files

By default, MATLAB Coder generates files in output folders based on your output type. For more information, see “Generated Files and Locations” on page 19-125.

Note Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

Specifying the Output Type Using the MATLAB Coder Project Interface

On the MATLAB Coder project **Build** tab, set **Output type** to one of the available output types:

- MEX Function (default)
- Instrumented MEX Function
- C/C++ Static Library
- C/C++ Dynamic Library
- C/C++ Executable

MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you switch the output type between MEX Function or Instrumented MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

Specifying the Output Type at the Command Line

Call `codegen` with the `-config` option. For example, suppose you have a primary function `foo` that takes no input parameters. The following table shows how to specify different output types when compiling `foo`. If a primary function has input parameters, you must specify these inputs. For more information, see “Primary Function Input Specification” on page 19-38.

Note C is the default language for code generation with MATLAB Coder. To generate C++ code, see “Specify a Language for Code Generation” on page 19-24.

To Generate:	Use This Command:
MEX function using the default code generation options	codegen foo
MEX function specifying code generation options	<pre> cfg = coder.config('mex'); % Set configuration parameters, for example, % enable a code generation report cfg.GenerateReport=true; % Call codegen, passing the configuration % object codegen -config cfg foo </pre>
Standalone C/C++ code and compile it to a library using the default code generation options	codegen -config:lib foo
Standalone C/C++ code and compile it to a library specifying code generation options	<pre> cfg = coder.config('lib'); % Set configuration parameters, for example, % enable a code generation report cfg.GenerateReport=true; % Call codegen, passing the configuration % object codegen -config cfg foo </pre>
Standalone C/C++ code and compile it to an executable using the default code generation options and specifying the main.c file at the command line	<pre> codegen -config:exe main.c foo </pre> <hr/> <p>Note You must specify a main function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 19-19</p> <hr/>

To Generate:	Use This Command:
Standalone C/C++ code and compile it to an executable specifying code generation options	<pre data-bbox="565 326 1213 517"> cfg = coder.config('exe'); % Set configuration parameters, for example, % specify main file cfg.CustomSource = 'main.c'; cfg.CustomInclude = 'c:\myfiles'; codegen -config cfg foo </pre> <hr data-bbox="565 578 1323 581"/> <p data-bbox="565 586 1323 682">Note You must specify a main function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 19-19</p>

Specify a Language for Code Generation

- “Specifying a Language for Code Generation in the Project Settings Dialog Box” on page 19-24
- “Specifying a Language for Code Generation at the Command Line” on page 19-25

MATLAB Coder can generate C or C++ libraries and executables. C is the default language. You can specify a language explicitly from the project settings dialog box or at the command line.

Specifying a Language for Code Generation in the Project Settings Dialog Box

- 1 Select a suitable compiler for your target language.
- 2 On the MATLAB Coder project **Build** tab, click the **More settings** link to open the **Project Settings** dialog box.
- 3 On the **All Settings** tab, in the **Advanced** group, set **Language** to C or C++.

Note If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Specifying a Language for Code Generation at the Command Line

- 1 Select a suitable compiler for your target language.
- 2 Create a configuration object for code generation. For example, for a library:

```
cfg = coder.config('lib');
```

- 3 Set the TargetLang property to 'C' or 'C++'. For example:

```
cfg.TargetLang = 'C++';
```

Note If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

See Also.

- “Working with Configuration Objects” on page 19-30
- “Setting Up the C/C++ Compiler”

Specify Output File Name

Specifying Output File Name in a Project

On the project **Build** tab, in the **Output File Name** field, enter the file name. The file name can include an existing path.

Note Do not put spaces in the file name.

By default, if the name of the first entry-point MATLAB file is *fcn1*, the output file name is:

- *fcn1* for C/C++ libraries and executables.
- *fcn1_mex* for MEX functions.

By default, MATLAB Coder generates files in the folder *project_folder/codegen/target/fcn1*:

- *project_folder* is your current project folder
- target is:
 - *mex* for MEX functions
 - *lib* for static C/C++ libraries
 - *dll* for dynamic C/C++ libraries
 - *exe* for C/C++ executables

Command Line Alternative

Use the `codegen` function `-o` option.

Specify Output File Locations

Specifying Output File Location in a Project

The output file location should not contain:

- Spaces, as this can lead to code generation failures in certain operating system configurations.
- Non 7-bit ASCII characters, such as Japanese characters.

1 On the project **Build** tab, click **More settings**.

2 In the Project Settings dialog box, click the **Paths** tab.

The default setting for the **Build Folder** field is A subfolder of the `project_folder`. By default, MATLAB Coder generates files in the folder `project_folder/codegen/target/fcn1`:

- `fcn1` is the name of the first entry-point file
- `target` is:
 - `mex` for MEX functions
 - `lib` for static C/C++ libraries
 - `dll` for dynamically-linked C/C++ libraries
 - `exe` for C/C++ executables

3 To change the output location, you can either:

- Set **Build Folder** to A subfolder of the current MATLAB working folder

MATLAB Coder generates files in the `MATLAB_working_folder/codegen/target/fcn1` folder

- Set **Build Folder** to Specified folder. In the **Build folder name** field, provide the path to the folder.

Command Line Alternative

Use the `codegen` function `-d` option.

Parameter Specification Methods

If you are using...	Use...	Details
A MATLAB Coder project	The Project Settings dialog box	“Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-28
<code>codegen</code> at the command line and want to specify a small number of parameters	Configuration objects	“Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on

If you are using...	Use...	Details
codegen in build scripts		page 19-29
codegen at the command line and want to specify a large number of parameters	Configuration object dialog boxes	“Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 19-34

Specify Build Configuration Parameters

- “Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-28
- “Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-29
- “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 19-34

You can specify build configuration parameters from the MATLAB Coder project settings dialog box, the command line, or configuration object dialog boxes.

Specifying Build Configuration Parameters in the Project Settings Dialog Box

- 1 On the MATLAB Coder project **Build** tab, click **More settings**.

The Project Settings dialog box opens. This dialog box provides the set of configuration parameters applicable to the output type that you select.

Note MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you switch the output type between MEX Function or Instrumented MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

- 2 Modify the parameters as required. For more information about parameters on a tab, click the **Help** button.

Changes to the parameter settings take place immediately.

- 3 After specifying the build parameters, you can generate code by clicking the **Build** button on the same tab.

Specifying Build Configuration Parameters at the Command Line Using Configuration Objects

Types of Configuration Objects. The `codegen` function uses configuration objects to customize your environment for code generation. The following table lists the available configuration objects.

Configuration Object	Description
<code>coder.CodeConfig</code>	If no Embedded Coder license is available or you disable use of the Embedded Coder license, specifies parameters for C/C++ library or executable generation. For more information, see the class reference information for <code>coder.CodeConfig</code> .
<code>coder.EmbeddedCodeConfig</code>	If an Embedded Coder license is available, specifies parameters for C/C++ library or executable generation. For more information, see the class reference information for <code>coder.EmbeddedCodeConfig</code> .
<code>coder.HardwareImplementation</code>	Specifies parameters of the target hardware implementation. If not specified, <code>codegen</code> generates code that is compatible with the MATLAB host computer. For more information, see the class reference information for <code>coder.HardwareImplementation</code> .
<code>coder.MexCodeConfig</code>	Specifies parameters for MEX code generation. For more information, see the class reference information for <code>coder.MexCodeConfig</code> .

Working with Configuration Objects. To use configuration objects to customize your environment for code generation:

- 1 In the MATLAB workspace, define configuration object variables, as described in “Creating Configuration Objects” on page 19-31.

For example, to generate a configuration object for C static library generation:

```
cfg = coder.config('lib');  
% Returns a coder.CodeConfig object if no  
% Embedded Coder license available.  
% Otherwise, returns a coder.EmbeddedCodeConfig object.
```

- 2 Modify the parameters of the configuration object as required, using one of these methods:
 - Interactive commands, as described in “Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-29
 - Dialog boxes, as described in “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 19-34
- 3 Call the `codegen` function with the `-config` option. Specify the configuration object as its argument.

The `-config` option instructs `codegen` to generate code for the target, based on the configuration property values. In the following example, `codegen` generates a C static library from a MATLAB function, `foo`, based on the parameters of a code generation configuration object, `cfg`, defined in the first step:

```
codegen -config cfg foo
```

The `-config` option specifies the type of output that you want to build — in this case, a C static library. For more information, see `codegen`.

Creating Configuration Objects. You can define a configuration object in the MATLAB workspace.

To Create...	Use a Command Such As...
MEX configuration object <code>coder.MexCodeConfig</code>	<pre>cfg = coder.config('mex');</pre>
Code generation configuration object for generating a standalone C/C++ library or executable <code>coder.CodeConfig</code>	<pre>% To generate a static library cfg = coder.config('lib'); % To generate a dynamic library cfg = coder.config('dll') % To generate an executable cfg = coder.config('exe');</pre> <hr/> <p>Note If an Embedded Coder license is available, creates a <code>coder.EmbeddedCodeConfig</code> object.</p> <p>If you use concurrent licenses, to disable check out of an Embedded Coder license, use one of the following commands:</p> <pre>cfg = coder.config('lib', 'ecoder', false) cfg = coder.config('dll', 'ecoder', false) cfg = coder.config('exe', 'ecoder', false)</pre> <hr/>

To Create...	Use a Command Such As...
<p>Code generation configuration object for generating a standalone C/C++ library or executable for an embedded target</p> <p><code>coder.EmbeddedCodeConfig</code></p>	<pre>% To generate a static library cfg = coder.config('lib'); % To generate a dynamic library cfg = coder.config('dll') % To generate an executable cfg = coder.config('exe');</pre> <hr/> <p>Note Requires an Embedded Coder license; otherwise creates a <code>coder.CodeConfig</code> object.</p>
<p>Hardware implementation configuration object</p> <p><code>coder.HardwareImplementation</code></p>	<pre>hwcfg = coder.HardwareImplementation</pre>

Each configuration object comes with a set of parameters, initialized to default values. You can change these settings, as described in “Modifying Configuration Objects at the Command Line Using Dot Notation” on page 19-32.

Modifying Configuration Objects at the Command Line Using Dot Notation. You can use dot notation to modify the value of one configuration object parameter at a time. Use this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties:

- To specify a main function during C/C++ code generation:

```
cfg = coder.config('exe');
cfg.CustomInclude = 'c:\myfiles';
cfg.CustomSource = 'main.c';
codegen -config cfg foo
```

- To automatically generate and launch code generation reports after generating a C/C++ static library:

```
cfg = coder.config('lib');  
cfg.GenerateReport= true;  
cfg.LaunchReport = true;  
codegen -config cfg foo
```

Saving Configuration Objects. Configuration objects do not automatically persist between MATLAB sessions. Use one of the following methods to preserve your settings:

Save a configuration object to a MAT-file and then load the MAT-file at your next session

For example, assume you create and customize a MEX configuration object `mexcfg` in the MATLAB workspace. To save the configuration object, at the MATLAB prompt, enter:

```
save mexcfg.mat mexcfg
```

The `save` command saves `mexcfg` to the file `mexcfg.mat` in the current folder.

To restore `mexcfg` in a new MATLAB session, at the MATLAB prompt, enter:

```
load mexcfg.mat
```

The `load` command loads the objects defined in `mexcfg.mat` to the MATLAB workspace.

Write a script that creates the configuration object and sets its properties.

You can rerun the script whenever you need to use the configuration object again.

Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes

- 1** Create a configuration object as described in “Creating Configuration Objects” on page 19-31.

For example, to create a `coder.MexCodeConfig` configuration object for MEX code generation:

```
mexcfg = coder.config('mex');
```

- 2** Open the property dialog box using one of these methods:
 - In the MATLAB workspace, double-click the configuration object variable.
 - At the MATLAB prompt, issue the `open` command, passing it the configuration object variable, as in this example:

```
open mexcfg
```

- 3** In the dialog box, modify configuration parameters as required, then click **Apply**.
- 4** Call the `codegen` function with the `-config` option. Specify the configuration object as its argument:

```
codegen -config mexcfg foo
```

The `-config` option specifies the type of output that you want to build. For more information, see `codegen`.

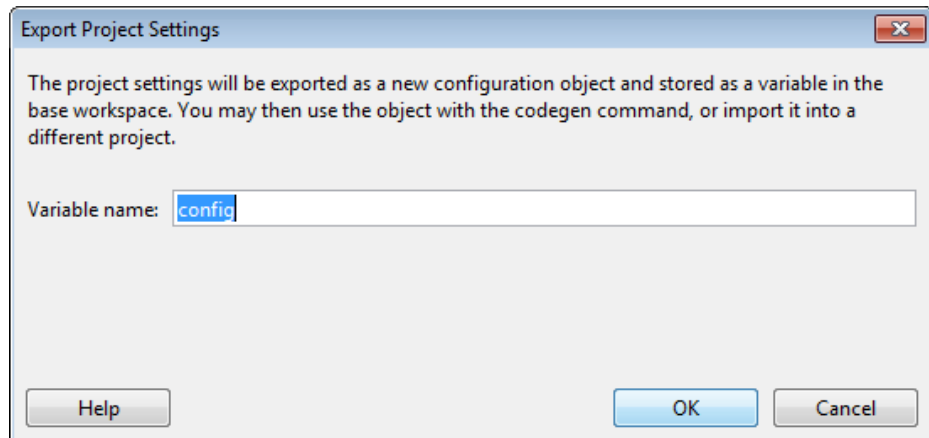
Share Build Configuration Settings

To share build configuration settings between multiple projects or between the project and command-line workflow, use the project `Export settings` and `Import settings` options.

Export Settings

To export the current project settings to a code generation configuration object stored in the base workspace:

- 1 In the top right corner of the project, click the **Actions** icon (⚙️) and select `Export settings`.
- 2 In the **Export Project Settings** dialog box, specify a name for the configuration object.



MATLAB Coder saves the project settings information in a configuration object with the specified name in the base workspace.

Project Output Type	Configuration Object
MEX Function	<code>coder.MexCodeConfig</code>
Instrumented MEX Function	

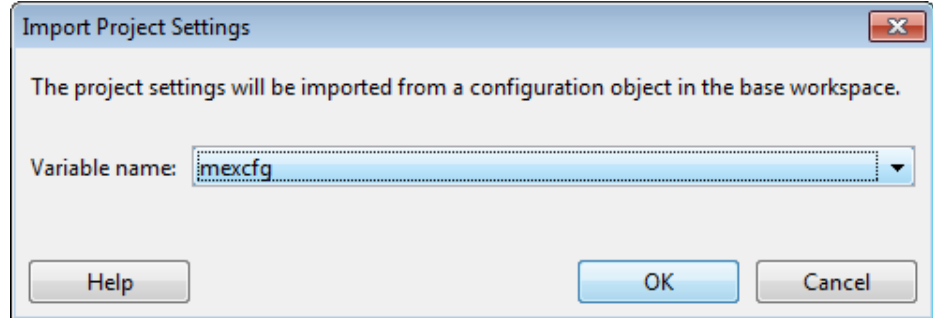
Project Output Type	Configuration Object
C/C++ Static Library	Without an Embedded Coder license:coder.CodeConfig
C/C++ Dynamic Library	With an Embedded Coder
C/C++ Executable	license:coder.EmbeddedCodeConfig

You can then either import these settings into another project or use it with the codegen function -config option to generate code at the command line.

Import Settings

To import the settings saved in a code generation configuration object stored in the base workspace:

- 1 In the top right corner of the project, click the **Actions** icon (⚙) and select **Import settings**.
- 2 In the **Import Project Settings** dialog box, select the configuration object that you want to use.



MATLAB Coder imports the settings saved in the configuration object and uses them as the current project settings.

Note When you import a coder.MexCodeConfig object, if the project output type is not already set to Instrumented MEX Function, the output type is set to MEX Function.

See Also

- “Build Setting Configuration” on page 19-21
- `coder.config`

Primary Function Input Specification

In this section...

“Why You Must Specify Input Properties” on page 19-38

“Properties to Specify” on page 19-38

“Rules for Specifying Properties of Primary Inputs” on page 19-42

“Methods for Defining Properties of Primary Inputs” on page 19-42

“Define Input Properties by Example at the Command Line” on page 19-43

“Specify Constant Inputs at the Command Line” on page 19-46

“Specify Variable-Size Inputs at the Command Line” on page 19-48

Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, MATLAB Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, to MATLAB Coder. If your primary function has no input parameters, MATLAB Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the primary function.

If you use the tilde (~) character to specify unused function inputs:

- In MATLAB Coder projects, if you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.
- When generating code with `codegen`, you must specify the type of these inputs using the `-args` option.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For...	Specify properties...				
	Class	Size	Complexity	numericity	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	<p>Specify properties for each field according to its class</p> <p>When a primary input is a structure, the code generation software treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition:</p> <ul style="list-style-type: none"> • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify numericity, and fimath. 				
Other inputs	✓	✓	✓		

Default Property Values

MATLAB Coder assigns the following default values for properties of primary function inputs.

Property	Default
class	double
size	scalar
complexity	real
numericity	No default
fimath	MATLAB default fimath object

Specifying Default Values for Structure Fields. In most cases, when you don't explicitly specify values for properties, MATLAB Coder uses defaults except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you might need to specify default values for properties of structure fields. For examples, see “Specifying Class and Size of Scalar Structure” on page 19-59 and “Specifying Class and Size of Structure Array” on page 19-60.

Specifying Default `fimath` Values for MEX Functions. MEX functions generated with MATLAB Coder use the default `fimath` value in effect at compile time. If you do not specify a default `fimath` value, MATLAB Coder uses the MATLAB default `fimath`. The MATLAB factory default has the following properties:

```
RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
CastBeforeSum: true
```

For more information, see “`fimath` for Sharing Arithmetic Rules”.

When running MEX functions that depend on the default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time warning, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` relies on the default `fimath` object in effect at compile time. At the MATLAB prompt, generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`:

```
codegen test
% codegen generates a MEX function, test_mex,
% in the current folder
```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```
test_mex
```

```
ans =
```

```
0
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 15

```

Now create a local MATLAB `fimath` value. so you no longer use the default setting:

```
F = fimath('RoundingMethod','Floor');
```

Finally, clear the MEX function from memory and rerun it:

```
clear test_mex
test_mex
```

The mismatch is detected and causes an error:

```
??? This function was generated with a different default
fimath than the current default.
```

```
Error in ==> test_mex
```

Supported Classes

The following table presents the class names supported by MATLAB Coder.

Class Name	Description
logical	Logical array of true and false values
char	Character array
int8	8-bit signed integer array
uint8	8-bit unsigned integer array

Class Name	Description
int16	16-bit signed integer array
uint16	16-bit unsigned integer array
int32	32-bit signed integer array
uint32	32-bit unsigned integer array
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
struct	Structure array
embedded.fi	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules.

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (`fi`), you must specify the input `numericType` and `fiMath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
“Specifying Properties of Primary Function Inputs in a Project” on page 16-7	<ul style="list-style-type: none"> • If you are working in a MATLAB Coder project, easy to use • Does not alter original MATLAB code 	<ul style="list-style-type: none"> • Not efficient for specifying memory-intensive inputs such as large structures and arrays

Method	Advantages	Disadvantages
<p>“Define Input Properties by Example at the Command Line” on page 19-43</p> <hr/> <p>Note If you define input properties programmatically in the MATLAB file, you cannot use this method</p>	<ul style="list-style-type: none"> • MATLAB Coder saves the definitions in the project file • Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a small number of primary inputs 	<ul style="list-style-type: none"> • Must be specified at the command line every time you invoke <code>codegen</code> (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
<p>“Define Input Properties Programmatically in the MATLAB File” on page 19-50</p>	<ul style="list-style-type: none"> • Integrated with MATLAB code; no need to redefine properties each time you invoke MATLAB Coder • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures 	<ul style="list-style-type: none"> • Uses complex syntax • MATLAB Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.

Define Input Properties by Example at the Command Line

- “Command Line Option `-args`” on page 19-44
- “Rules for Using the `-args` Option” on page 19-44

- “Specifying Properties of Primary Inputs by Example at the Command Line” on page 19-45
- “Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line” on page 19-45

Command Line Option -args

The `codegen` function provides a command-line option `-args` for specifying the properties of primary (entry-point) function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you generate code for the MATLAB function with `codegen`. If you have a test function or script that calls the entry-point MATLAB function with the required types, you can use `coder.getArgTypes` to determine the types of the function inputs. `coder.getArgTypes` returns a cell array of `coder.Type` objects that you can pass to `codegen` using the `-args` option. For more information, see the `coder.getArgTypes` function reference information.

See “Specifying General Properties of Primary Inputs” on page 19-58 for `codegen`.

Rules for Using the -args Option

When using the `-args` command-line option to define properties by example, follow these rules:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Note If you specify an empty cell array with the `-args` option, `codegen` interprets this to mean that the function takes no inputs; a compile-time error occurs if the function does have inputs.

Specifying Properties of Primary Inputs by Example at the Command Line

Consider a MATLAB function that adds its two inputs:

```
function y = mcf(u,v)
%#codegen
y = u + v;
```

The following examples show how to specify different properties of the primary inputs *u* and *v* by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

```
codegen mcf -args {0,0}
```

- Use a literal cell array of constants to specify that input *u* is an unsigned 16-bit, 1-by-4 vector and input *v* is a scalar double:

```
codegen mcf -args {zeros(1,4,'uint16'),0}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = uint8([1;2;3;4])
b = uint8([5;6;7;8])
ex = {a,b}
codegen mcf -args ex
```

Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line

To generate a MEX function or C/C++ code for fixed-point MATLAB code, you must install Fixed-Point Designer software.

Consider a MATLAB function that calculates the square root of a fixed-point number:

```
%#codegen
function y = sqrtfi(x)
y = sqrt(x);
```

To specify the properties of the primary fixed-point input *x* by example on the MATLAB command line, follow these steps:

- 1 Define the `numericType` properties for *x*, as in this example:

```
T = numericType('WordLength',32,...
               'FractionLength',23,...
               'Signed',true);
```

- 2 Define the `fimath` properties for *x*, as in this example:

```
F = fimath('SumMode','SpecifyPrecision',...
          'SumWordLength',32,...
          'SumFractionLength',23,...
          'ProductMode','SpecifyPrecision',...
          'ProductWordLength',32,...
          'ProductFractionLength',23);
```

- 3 Create a fixed-point variable with the `numericType` and `fimath` properties you just defined, as in this example:

```
myeg = { fi(4.0,T,F) };
```

- 4 Compile the function `sqrtfi` using the `codegen` command, passing the variable `myeg` as the argument to the `-args` option, as in this example:

```
codegen sqrtfi -args myeg;
```

Specify Constant Inputs at the Command Line

In cases where you know your primary inputs will not change at run time, it is more efficient to specify them as constant values than as variables to eliminate unnecessary overhead in generated code. Common uses of constant inputs are for flags that control how an algorithm executes and values that specify the sizes or types of data.

You can define inputs to be constants using the `-args` command-line option with a `coder.Constant` object, as in this example:

```
-args {coder.Constant(constant_input)}
```


This expression specifies that an input will be a constant with the size, class, complexity, and value of *constant_input*.

Calling Functions with Constant Inputs

codegen compiles constant function inputs into the generated code. As a result, the MEX function signature differs from the MATLAB function signature. At run time you supply the constant argument to the MATLAB function, but not to the MEX function.

For example, consider the following function identity which copies its input to its output:

```
function y = identity(u) %#codegen
y = u;
```

To generate a MEX function `identity_mex` with a constant input, at the MATLAB prompt, type the following command:

```
codegen identity -args {coder.Constant(42)}
```

To run the MATLAB function, supply the constant argument:

```
identity(42)
```

You get the following result:

```
ans =

    42
```

Now, try running the MEX function with this command:

```
identity_mex
```

You should get the same answer.

Specifying a Structure as a Constant Input

Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
y = zeros(p.rows,p.cols) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
codegen rowcol -args {0,coder.Constant(tmp)}
```

Specify Variable-Size Inputs at the Command Line

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. You can define inputs to have one or more variable-size dimensions — and specify their upper bounds — using the `-args` option and `coder.typeof` function:

```
-args {coder.typeof(example_value, size_vector, variable_dims)}
```

Specifies a variable-size input with:

- Same class and complexity as *example_value*
- Same size and upper bounds as *size_vector*
- Variable dimensions specified by *variable_dims*

When you enable dynamic memory allocation, you can specify `Inf` in the size vector for dimensions with unknown upper bounds at compile time.

When *variable_dims* is a scalar, it is applied to all the dimensions, with the following exceptions:

- If the dimension is 1 or 0, which are fixed.
- If the dimension is unbounded, which is always variable size.

For more information, see `coder.typeof` and “Generate Code for Variable-Size Data” on page 19-98.

Specifying a Variable-Size Vector Input

- 1 Write a function that computes the average of every n elements of a vector A and stores them in a vector B :

```
function B = nway(A,n) %#codegen
% Compute average of every N elements of A and put them in B.

coder.extrinsic('error');
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
        B(i) = mean(A(k + (0:n-1)));
        k = k + n;
    end
else
    B = zeros(1,0);
    error('n <= 0 or does not divide number of elements evenly');
end
```

- 2 Specify the first input A as a vector of double values. Its first dimension stays fixed in size and its second dimension can grow to an upper bound of 100. Specify the second input n as a double scalar.

```
codegen -report nway -args {coder.typeof(0,[1 100],1),1}
```

- 3 As an alternative, assign the `coder.typeof` expression to a MATLAB variable, then pass the variable as an argument to `-args`:

```
vareg = coder.typeof(0,[1 100],1)
codegen -report nway -args {vareg, 0}
```

Define Input Properties Programmatically in the MATLAB File

With MATLAB Coder, you use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

In this section...

- “How to Use `assert` with MATLAB® Coder™” on page 19-50
- “Rules for Using `assert` Function” on page 19-57
- “Specifying General Properties of Primary Inputs” on page 19-58
- “Specifying Properties of Primary Fixed-Point Inputs” on page 19-59
- “Specifying Class and Size of Scalar Structure” on page 19-59
- “Specifying Class and Size of Structure Array” on page 19-60

How to Use `assert` with MATLAB Coder

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

You must use one of the following methods when specifying input properties using the `assert` function. Use the exact syntax that is provided; do not modify it.

- “Specify Any Class” on page 19-51
- “Specify `fi` Class” on page 19-51
- “Specify Structure Class” on page 19-52
- “Specify Fixed Size” on page 19-52
- “Specify Scalar Size” on page 19-53
- “Specify Upper Bounds for Variable-Size Inputs” on page 19-53
- “Specify Inputs with Fixed- and Variable-Size Dimensions” on page 19-53
- “Specify Size of Individual Dimensions” on page 19-54
- “Specify Real Input” on page 19-55

- “Specify Complex Input” on page 19-55
- “Specify numerictype of Fixed-Point Input” on page 19-55
- “Specify fimath of Fixed-Point Input” on page 19-56
- “Specify Multiple Properties of Input” on page 19-56

Specify Any Class

```
assert ( isa ( param, 'class_name' ) )
```

Sets the input parameter *param* to the MATLAB class *class_name*. For example, to set the class of input *U* to a 32-bit signed integer, call:

```
...
assert(isa(U, 'int32'));
...
```

If you set the class of an input parameter to *fi*, you must also set its *numerictype*, see “Specify numerictype of Fixed-Point Input” on page 19-55. You can also set its *fimath* properties, see “Specify fimath of Fixed-Point Input” on page 19-56. If you do not set the *fimath* properties, *codegen* uses the MATLAB default *fimath* value.

If you set the class of an input parameter to *struct*, you must specify the properties of all fields in the order that they appear in the structure definition.

Specify fi Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class *fi* (fixed-point numeric object). For example, to set the class of input *U* to *fi*, call:

```
...
assert(isfi(U));
...
```

or

```
...  
assert(isa(U, 'embedded.fi'));  
...
```

If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see “Specify `numerictype` of Fixed-Point Input” on page 19-55. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 19-56. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Structure Class

```
assert ( isstruct ( param ) )  
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter `param` to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...  
assert(isstruct(U));  
...  
  
or  
  
...  
assert(isa(U, 'struct'));  
...
```

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Fixed Size

```
assert ( all ( size (param) == [dims] ) )
```

Sets the input parameter `param` to the size specified by dimensions `dims`. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size

```
assert ( isscalar (param ) )
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. To set the size of input *U* to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
assert(all(size(U)== [1]));
...
```

Specify Upper Bounds for Variable-Size Inputs

```
assert ( all(size(param)<=[N0 N1 ...]) );
assert ( all(size(param)<[N0 N1 ...]) );
```

Sets the upper-bound size of each dimension of input parameter *param*. To set the upper-bound size of input *U* to be less than or equal to a 3-by-2 matrix, call:

```
assert(all(size(U)<=[3 2]));
```

Note You can also specify upper bounds for variable-size inputs using `coder. varsize`.

Specify Inputs with Fixed- and Variable-Size Dimensions

```
assert ( all(size(param)>=[M0 M1 ...]) );
```

```
assert ( all(size(param)<=[N0 N1 ...]));
```

When you use `assert(all(size(param)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:

- You must also specify an upper-bound size for each dimension of the input parameter.
- For each dimension, k , the lower-bound M_k must be less than or equal to the upper-bound N_k .
- To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
- Bounds must be non-negative.

To fix the size of the first dimension of input `U` to 3 and set the second dimension as variable size with upper-bound of 2, call:

```
assert(all(size(U)>=[3 0]));  
assert(all(size(U)<=[3 2]));
```

Specify Size of Individual Dimensions

```
assert (size(param, k)==Nk);  
assert (size(param, k)<=Nk);  
assert (size(param, k)<Nk);
```

You can specify individual dimensions as well as specifying all dimensions simultaneously or instead of specifying all dimensions simultaneously. The following rules apply:

- You must specify the size of each dimension at least once.
- The last dimension specification takes precedence over earlier specifications.

Sets the upper-bound size of dimension k of input parameter `param`. To set the upper-bound size of the first dimension of input `U` to 3, call:

```
assert(size(U,1)<=3)
```


To fix the size of the second dimension of input *U* to 2, call:

```
assert(size(U,2)==2)
```

Specify Real Input

```
assert ( isreal (param ) )
```

Specifies that the input parameter *param* is real. To specify that input *U* is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *param* is complex. To specify that input *U* is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the `numerictype` properties of `fi` input parameter *fiparam* to the `numerictype` object *T*. For example, to specify the `numerictype` property of fixed-point input *U* as a signed `numerictype` object *T* with 32-bit word length and 30-bit fraction length, use the following code:

```
 %#codegen
...
% Define the numerictype object.
```

```
T = numericity(1, 32, 30);

% Set the numericity property of input U to T.
assert(isequal(numericity(U),T));
...
```

Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter `fiparam` to the `fimath` object `F`. For example, to specify the `fimath` property of fixed-point input `U` so that it saturates on integer overflow, use the following code:

```
 %#codegen
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

If you do not specify the `fimath` properties using `assert`, `codegen` uses the MATLAB default `fimath` value.

Specify Multiple Properties of Input

```
assert ( function1 ( params ) &&
         function2 ( params ) &&
         function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
 %#codegen
...
assert(isa(U,'double') &&
       ~isreal(U) &&
```

```

    all(size(U) == [3 3]) &&
    isa(V, 'uint16');
...

```

Rules for Using assert Function

When using the `assert` function to specify the properties of primary function inputs, follow these rules:

- Call `assert` functions at the beginning of the primary function, before control-flow operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- Use the `assert` function with MATLAB Coder only for specifying properties of primary function inputs before converting your MATLAB code to C/C++ code.
- If you set the class of an input parameter to `fi`, you must also set its `numericType`. See “Specify `numericType` of Fixed-Point Input” on page 19-55. You can also set its `fimath` properties. See “Specify `fimath` of Fixed-Point Input” on page 19-56. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order that they appear in the structure definition.
- When you use `assert(all(size(param) >= [M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:
 - You must also specify an upper-bound size for each dimension of the input parameter.
 - For each dimension, k , the lower-bound M_k must be less than or equal to the upper-bound N_k .
 - To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
 - Bounds must be non-negative.
- If you specify individual dimensions, the following rules apply:
 - You must specify the size of each dimension at least once.

- The last dimension specification takes precedence over earlier specifications.

Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `mcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs:

Input	Property	Value
pennywhistle	class	int16
	size	220500-by-1 vector
	complexity	real (by default)
win	class	double
	size	1024-by-1 vector
	complexity	real (by default)

```

%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...

```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands:

```

%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...

```

Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Designer software.

In the following example, the primary MATLAB function `mcsqrtfi` takes one fixed-point input `x`. The code specifies the following properties for this input.

Property	Value
<code>class</code>	<code>fi</code>
<code>numericitype</code>	numericitype object T, as specified in the primary function
<code>fimath</code>	fimath object F, as specified in the primary function
<code>size</code>	scalar
<code>complexity</code>	real (by default)

```
function y = mcsqrtfi(x) %#codegen
T = numericitype('WordLength',32,'FractionLength',23,...
    'Signed',true);
F = fimath('SumMode','SpecifyPrecision',...
    'SumWordLength',32,'SumFractionLength',23,...
    'ProductMode','SpecifyPrecision',...
    'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numericitype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Specifying Class and Size of Scalar Structure

Assume you have defined `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

Here is code that specifies the class and size of `S` and its fields when passed as an input to your MATLAB function:

```
function y = fcn(S) %#codegen
```

```
% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8'));
...
```

In most cases, when you don't explicitly specify values for properties, MATLAB Coder uses defaults — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of a structure field

Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined `S` as the following 2-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input `S` using the first element of the array:

```
##codegen
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [2 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of all fields.

Speed Up Compilation

In this section...
“Generate Code Only” on page 19-61
“Disable Compiler Optimization” on page 19-61

Generate Code Only

If you select this option, MATLAB Coder does not invoke the make command or generate compiled object code. When you want to iterate rapidly between modifying MATLAB code and generating C/C++ code and you want to inspect the generated code, this option saves you time during the development cycle .

In the Project Interface

On the project **Build** tab, select **Generate code only**.

At the Command Line

Use the `codegen -c` option to only generate code without invoking the make command. For example, to generate code only for a function, `foo`, that takes one single, scalar input:

```
codegen -c foo -args {single(0)}
```

For more information and a complete list of compilation options, see `codegen`.

Disable Compiler Optimization

Turning compiler optimizations off shortens compile time, but increases run time.

In the Project Interface

- 1 On the MATLAB Coder project **Build** tab, verify that the **Output type** is C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable.

- 2 On the **Build** tab, click the **More settings** link.

- 3** In the **Project Settings** dialog box **All Settings** tab, under **Advanced**, set **Compiler optimization level** to **Off**.

At the Command Line

- 1** Create a code generation configuration object for C/C++ library or executable. For example, for a static library:

```
cfg = coder.config('lib');
```

- 2** Set the `CCompilerOptimization` to `Off`.

```
cfg.CCompilerOptimization='Off';
```


Code Optimization

In this section...

“Unroll for-loops” on page 19-63

“Inline Code” on page 19-65

“Eliminate Redundant Copies of Function Inputs (A=foo(A))” on page 19-66

“Rewrite Logical Array Indexing as a Loop” on page 19-68

Unroll for-loops

Unrolling for-loops eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant.

You can also force loop unrolling for individual functions by wrapping the loop header in a `coder.unroll` function. For more information, see `coder.unroll`.

Limiting Copying the Body of a for-loop in Generated Code

To limit the number of times to copy the body of a for-loop in generated code:

- 1 Write a MATLAB function `getrand(n)` that uses a for-loop to generate a vector of length `n` and assign random numbers to specific elements. Add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the for-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Calling getrand 8 times triggers unroll
y1 = getrand(8);
% Calling getrand 50 times does not trigger unroll
y2 = getrand(50);
```

```
function y = getrand(n)
% Turn off inlining to make
% generated code easier to read
```

```
coder.inline('never');

% Set flag variable dounroll to repeat loop body
% only for fewer than 10 iterations
dounroll = n < 10;
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
for i = coder.unroll(1:2:n, dounroll)
    if (i > 2) && (i < n-2)
        y(i) = rand();
    end;
end;
% Loop body ends
```

- 2** In the default output folder, `codegen/lib/test_unroll`, generate C static library code for `test_unroll` :

```
codegen -config:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the for-loop (unrolls the loop) because the number of iterations is less than 10:

```
static void m_getrand(real_T y[8])
{
    int32_T i0;
    for(i0 = 0; i0 < 8; i0++) {
        y[i0] = 0.0;
    }
    /* Loop body begins */
    y[2] = m_rand();
    y[4] = m_rand();
    /* Loop body ends */
}
```

The generated C code for `getrand(50)` does not unroll the for-loop because the number of iterations is greater than 10:

```
static void m_b_getrand(real_T y[50])
```

```

{
  int32_T i;
  for(i = 0; i < 50; i++) {
    y[i] = 0.0;
  }
  /* Loop body begins */
  for(i = 0; i < 50; i += 2) {
    if((i + 1 > 2) && (i + 1 < 48)) {
      y[i] = m_rand();
    }
  }
  /* Loop body ends */
}

```

Inline Code

MATLAB uses internal heuristics to determine whether or not to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. For more information, see `coder.inline`.

Preventing Function Inlining

In this example, function `foo` is not inlined in the generated code:

```

function y = foo(x)
  coder.inline('never');
  y = x;
end

```

Using Inlining in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```

function y = inline_division(dividend, divisor)

```

```
% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
    coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
    coder.inline('never');
end

if any(divisor == 0)
    error('Can not divide by 0');
end

y = dividend / divisor;
```

Eliminate Redundant Copies of Function Inputs (A=foo(A))

You can reduce the number of copies in your generated code by writing functions that use the same variable as both an input and an output. For example:

```
function A = foo( A, B ) %#codegen
A = A * B;
end
```

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, MATLAB passes the variable by reference in the generated code instead of redundantly copying the input to a temporary variable. In the preceding example, input A is passed by reference in the generated code because it also acts as an output for function foo:

```
...
/* Function Definitions */
void foo(real_T *A, real_T B)
{
    *A *= B;
}
...
```

The reference parameter optimization reduces memory usage and execution time, especially when the variable passed by reference is a large data structure. To achieve these benefits at the call site, call the function with the same variable as both input and output.

By contrast, suppose you rewrite function `foo` without the optimization:

```
function y = foo2( A, B ) %#codegen
y = A * B;
end
```

MATLAB generates code that passes the inputs by value and returns the value of the output:

```
...
/* Function Definitions */
real_T foo2(real_T A, real_T B)
{
    return A * B;
}
...
```

In some cases, the output of the function cannot be a modified version of its inputs. If you do not use the inputs later in the function, you can modify your code to operate on the inputs instead of on a copy of the inputs. One method is to create additional return values for the function. For example, consider the code:

```
function y1=foo(u1) %#codegen
    x1=u1+1;
    y1=bar(x1);
end

function y2=bar(u2)
    % Since foo does not use x1 later in the function,
    % it would be optimal to do this operation in place
    x2=u2.*2;
    % The change in dimensions in the following code
    % means that it cannot be done in place
    y2=[x2,x2];
end
```

You can modify this code to eliminate redundant copies. The changes are highlighted in bold.

```
function y1=foo(u1) %#codegen
    u1=u1+1;
    [y1, u1]=bar(u1);
end
```

```
function [y2, u2]=bar(u2)
    u2=u2.*2;
    % The change in dimensions in the following code
    % still means that it cannot be done in place
    y2=[u2,u2];
end
```

Rewrite Logical Array Indexing as a Loop

Rewriting logical array indexing as a loop can optimize the generated C/C++ code for both speed and readability. For example, the MATLAB function, `foo`, uses logical array indexing.

```
function x = foo(x,N) %#codegen
assert(all(size(x) == [1 100]))
x(x>N) = N;
```

The generated C code for this function is not very efficient. Rewrite the MATLAB code to use a loop instead of logical indexing:

```
function x = foo_rewrite(x,N) %#codegen
assert(all(size(x) == [1 100]))
for ii=1:numel(x)
    if x(ii) > N
        x(ii) = N;
    end
end
```

Paths and File Infrastructure Setup

In this section...

“Compile Path Search Order” on page 19-69

“Specifying Folders to Search for Custom Code” on page 19-69

“Naming Conventions” on page 19-70

Compile Path Search Order

MATLAB Coder resolves MATLAB functions by searching first on the *code generation path* and then on the MATLAB path. The code generation path contains the current folder and the code generation libraries. By default, unless MATLAB Coder determines that a function should be extrinsic or you explicitly declare the function to be extrinsic, MATLAB Coder tries to compile and generate code for functions it finds on the path. MATLAB Coder does not compile extrinsic functions, but rather dispatches them to the MATLAB interpreter for execution. See “Resolution of Function Calls in MATLAB Generated Code” on page 13-2.

Specifying Folders to Search for Custom Code

If you want to integrate custom code — such as source, header, and library files — with the generated code, you can specify additional folder to search. The following table describes how to specify these search paths. The path should not contain spaces, as this can lead to code generation failures in certain operating system configurations. If the path contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not be able to find files on this path.

To specify additional folders:	Do this:
Using the MATLAB Coder interface	On the MATLAB Coder project Build tab: <ol style="list-style-type: none"> 1 Click the More settings link. 2 In the Project Settings dialog box, click the Paths tab. 3 For the Search paths field, either browse to add a folder to the search path or enter the full path. The search path must not contain spaces.
At the command line	Use the <code>codegen</code> function <code>-I</code> option.

Naming Conventions

MATLAB Coder enforces naming conventions for MATLAB functions and generated files.

- “Reserved Prefixes” on page 19-70
- “Reserved Keywords” on page 19-70
- “Conventions for Naming Generated files” on page 19-74

Reserved Prefixes

MATLAB Coder reserves the prefix `eml` for global C/C++ functions and variables in generated code. For example, MATLAB for code generation run-time library function names begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C/C++ functions or primary MATLAB functions with the prefix `eml`.

Reserved Keywords

- “C Reserved Keywords” on page 19-71
- “C++ Reserved Keywords” on page 19-71
- “Reserved Keywords for Code Generation” on page 19-72

- “MATLAB® Coder™ Code Replacement Library Keywords” on page 19-72

MATLAB Coder software reserves certain words for its own use as keywords of the generated code language. MATLAB Coder keywords are reserved for use internal to MATLAB Coder software and should not be used in MATLAB code as identifiers or function names. C reserved keywords should also not be used in MATLAB code as identifiers or function names. If your MATLAB code contains reserved keywords, the code generation build does not complete and an error message is displayed. To address this error, modify your code to use identifiers or names that are not reserved.

If you are generating C++ code using the MATLAB Coder software, in addition, your MATLAB code must not contain the “C++ Reserved Keywords” on page 19-71.

C Reserved Keywords.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C++ Reserved Keywords.

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual

explicit	operator	this	wchar_t
export	private	throw	

Reserved Keywords for Code Generation.

abs	fortran	localZCE	rtNaN
asm	HAVESTDIO	localZCSV	SeedFileBuffer
bool	id_t	matrix	SeedFileBufferLen
boolean_T	int_T	MODEL	single
byte_T	int8_T	MT	TID01EQ
char_T	int16_T	NCSTATES	time_T
cint8_T	int32_T	NULL	true
cint16_T	int64_T	NUMST	TRUE
cint32_T	INTEGER_CODE	pointer_T	uint_T
creal_T	LINK_DATA_BUFFER_SIZE	PROFILING_ENABLED	uint8_T
creal32_T	LINK_DATA_STREAM	PROFILING_NUM_SAMPLES	uint16_T
creal64_T	localB	real_T	uint32_T
cuint8_T	localC	real32_T	uint64_T
cuint16_T	localDWork	real64_T	UNUSED_PARAMETER
cuint32_T	localP	RT	USE_RTMODEL
ERT	localX	RT_MALLOC	VCAST_FLUSH_DATA
false	localXdis	rtInf	vector
FALSE	localXdot	rtMinusInf	

MATLAB Coder Code Replacement Library Keywords. The list of code replacement library (CRL) reserved keywords for your development environment varies depending on which CRLs currently are registered. Beyond the default ANSI, ISO, and GNU® CRLs provided with MATLAB Coder software, additional CRLs might be registered and available for use if you have installed other products that provide CRLs (for example, a target product), or if you have used Embedded Coder APIs to create and register custom CRLs.

To generate a list of reserved keywords for the CRLs currently registered in your environment, use the following MATLAB function:

```
crl_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers()
```

This function returns an array of CRL keyword strings. Specifying the return argument is optional.

Note To list the CRLs currently registered in your environment, use the MATLAB command `RTW.viewTf1`.

To generate a list of reserved keywords for the CRL that you are using to generate code, call the function passing the name of the CRL as displayed in the **Code replacement library** menu on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. For example,

```
crl_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU99 (GNU)')
```

Here is a partial example of the function output:

```
>> crl_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU99 (GNU)')

crl_ids =

    'exp10'
    'exp10f'
    'acosf'
    'acoshf'
    'asinf'
    'asinhf'
    'atanf'
    'atanhf'
    ...
    'rt_lu_cplx'
    'rt_lu_cplx_sgl'
    'rt_lu_real'
    'rt_lu_real_sgl'
    'rt_mod_boolean'
    'rt_rem_boolean'
```

```
'strcpy'
'utAssert'
```

Note Some of the returned keyword strings appear with the suffix \$N, for example, 'rt_atan2\$N'. \$N expands into the suffix _snf only if nonfinite numbers are supported. For example, 'rt_atan2\$N' represents 'rt_atan2_snf' if nonfinite numbers are supported and 'rt_atan2' if nonfinite numbers are not supported. As a precaution, you should treat both forms of the keyword as reserved.

Conventions for Naming Generated files

The following table describes how MATLAB Coder names generated files. MATLAB Coder follows MATLAB conventions by providing platform-specific extensions for MEX files.

Platform	MEX File Extension	MATLAB Coder Library Extension	MATLAB Coder Executable Extension
Linus Torvalds' Linux (32-bit)	.mexglx	.a	None
Linux x86-64	.mexa64	.a	None
Microsoft Windows (32-bit)	.mexw32	.lib	.exe
Windows x64	.mexw64	.lib	.exe

Generate Code for Multiple Entry-Point Functions

In this section...

“Advantages of Generating Code for More Than One Entry-Point Function” on page 19-75

“Generating Code for More Than One Entry-Point Function Using the Project Interface” on page 19-75

“Generating Code for More Than One Entry-Point Function at the Command Line” on page 19-78

“How to Call an Entry-Point Function in a MEX Function” on page 19-79

“How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code” on page 19-80

Advantages of Generating Code for More Than One Entry-Point Function

Generating a single C/C++ library for more than one entry-point MATLAB function allows you to:

- Create C/C++ libraries containing multiple, compiled MATLAB files to integrate with larger C/C++ applications.
- Share code efficiently between library functions.
- Communicate between library functions using shared memory.

Generating a MEX function for more than one entry-point function allows you to validate entry-point interactions in MATLAB before creating a C/C++ library.

Generating Code for More Than One Entry-Point Function Using the Project Interface

In the project, in the **Entry-Point Files** pane on the **Overview** tab, click the **Add files** link. Browse to the file that you want to add. Repeat this action for each entry-point file.

By default, MATLAB Coder:

- Lists the entry-point files alphabetically.
- Generates a MEX function in the current folder. MATLAB Coder names the MEX function , *fun_1_mex*. *fun_1* is the name of the first entry-point function.
- Stores generated files in the subfolder `codegen/mex/`*fun_1/*.

Generating a MEX Function with Two Entry-Point Functions Using the Project Interface

Generate a MEX function with two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector.

- 1** In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

- 2** In the same folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

- 3** In the same folder, set up a MATLAB Coder project.

- a** At the MATLAB command line, enter:

```
coder -new ep.prj
```

By default, the project opens in the MATLAB workspace on the right side.

- b** On the project **Overview** tab, click the `Add files` link. Browse to the file `ep1.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab, and the input is undefined.

- c** Define the type of input `u`.

- i** On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select `single`.
- ii** From the list of size options, select `1 x 1` to specify that `u` is a scalar.
- d** On the project **Overview** tab, click the `Add files` link. Browse to the file `ep2.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab, and the inputs are undefined.

- e** Define the type of input `u`.
 - iii** On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select `double`.
 - iv** From the list of size options, select `1 x 1` to specify that `u` is a scalar.
 - f** Repeat the previous step for input `v`, setting the **Size** to `2x1`.
- 4** In the MATLAB Coder project, click the **Build** tab.

By default, the **Output type** is `MEX function` and the **Output file name** is `ep1_mex`.

- 5** On this tab, click the **Build** button to generate a MEX function using the default project settings.

MATLAB Coder builds the project and, by default, generates a MEX function, `ep1_mex`, in the current folder. MATLAB Coder also generates other supporting files in a subfolder called `codegen/mex/ep1_mex`. MATLAB Coder uses the name of the MATLAB function as the root name for the generated files and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 19-70.

You can now test your MEX function in MATLAB. For more information, see “How to Call an Entry-Point Function in a MEX Function” on page 19-79.

Generating a C Static Library with Two Entry-Point Functions Using the Project Interface

You can generate a C static library with two entry-point functions, `ep1` and `ep2`, following the same project setup steps that you use to generate a MEX function. (See [Generating a MEX Function with Two Entry-Point Functions](#))

Using the Project Interface on page 19-76.) When you build the project, set the **Output type** to C/C++ Static Library.

MATLAB Coder builds the project and generates a C library, `ep1`, and supporting files in the default folder, `codegen/lib/ep1`.

You can now test your library. For more information, see “How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code” on page 19-80.

Generating Code for More Than One Entry-Point Function at the Command Line

To generate code for more than one entry-point function, use the following syntax, where `global_options` applies to functions, `fun_1` through `fun_n`, and `options_n` applies only to the preceding function `fun_n`.

```
codegen -global_options fun_1 -options_1 ... fun_n -options_n
```

By default, `codegen`:

- Generates a MEX function in the current folder. `codegen` names the MEX function, `fun_1_mex`. `fun_1` is the name of the first entry-point function.
- Stores generated files in the subfolder `codegen/mex/fun_1/`.

If you specify an output file name, `out_fun`, using the `-o` option, `codegen` stores the generated files in the subfolder `codegen/mex/out_fun/`. For more information on setting build options at the command line, see `codegen`.

Generating a MEX Function with Two Entry-Point Functions at the Command Line

Generate a MEX function with two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Using the `-o` option, name the generated MEX function `sharedmex`.

```
codegen -o sharedmex ep1 -args single(0) ep2 -args { 0, zeros(1,1024) }
```


codegen generates a MEX function named `sharedmex` in the current folder and stores generated files in the subfolder `codegen/mex/sharedmex`.

Note By default, codegen generates a MEX function named `ep1_mex` in the subfolder, `codegen/mex/ep1`.

Generating a C/C++ Static Library with Two Entry-Point Functions at the Command Line

Generate standalone C/C++ code and compile it to a library for two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Use the `-config:lib` option to specify that the target is a library. Using the `-o` option, name the generated library function `sharedlib`.

```
codegen -config:lib -o sharedlib ep1 -args single(0) ep2 ...
        -args { 0, zeros(1,1024) }
```

codegen generates C/C++ library code in the `codegen\lib\sharedlib` folder.

Note By default, codegen generates a library function named `ep1` in the subfolder, `codegen/lib/ep1`.

For information on viewing entry-point functions in the code generation report, see “Code Generation Reports” on page 19-176.

How to Call an Entry-Point Function in a MEX Function

To call an entry-point function in a MEX function that has more than one entry point, use this syntax:

```
MEX_Function('entry_point_function_name',
             ... entry_point_function_param1,
             ... , entry_point_function_paramn)
```

Calling an Entry-Point Function in a MEX Function

Consider a MEX function, `sharedmex`, that has entry-point functions `ep1` and `ep2`. Entry-point function `ep1` takes one single scalar input and `ep2` takes two inputs, a double scalar and a double vector.

To call `ep1` with an input parameter `u`, enter:

```
sharedmex('ep1', u)
```

To call `ep2` with input parameters `u` and `v`, enter:

```
sharedmex('ep2', u, v)
```

How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code

To call an entry-point function in a C/C++ library function from C/C++ code, write a `main` function in C/C++ that:

- Includes the generated header files, which contain the function prototypes for the entry-point functions.
- Calls the `initialize` function before calling the entry-point functions for the first time.
- Calls the `terminate` function after calling the entry-point functions for the last time.
- Configures your target to integrate this custom C/C++ main function with your generated code, as described in “Custom C/C++ Code Integration” on page 21-12.
- Generates the C/C++ executable using `codegen`.

See the example, “Call a C Static Library Function from C Code” on page 21-2.

Generate Code for Global Data

In this section...

“Workflow” on page 19-81

“Declare Global Variables” on page 19-81

“Define Global Data” on page 19-82

“Synchronizing Global Data with MATLAB” on page 19-83

“Limitations of Using Global Data” on page 19-87

Workflow

To generate C/C++ code from MATLAB code that uses global data:

- 1 Declare the variables as global in your code.
- 2 Before using the global data, define and initialize it.

For more information, see “Define Global Data” on page 19-82.
- 3 Generate code from the MATLAB Coder project interface or using codegen.

If you use global data, you must also specify whether you want to synchronize this data between MATLAB and the generated MEX function. For more information, see “Synchronizing Global Data with MATLAB” on page 19-83.

Declare Global Variables

When using global data, you must first declare the global variables in your MATLAB code. Consider the `use_globals` function that uses two global variables `AR` and `B`:

```
function y = use_globals(u)
%#codegen
% Turn off inlining to make
% generated code easier to read
coder.inline('never');
% Declare AR and B as global variables
global AR;
```

```
global B;  
AR(1) = u + B(1);  
y = AR * 2;
```

Define Global Data

You can define global data either in the MATLAB global workspace, in a MATLAB Coder project, or at the command line. If you do not initialize global data in a project or at the command line, MATLAB Coder looks for the variable in the MATLAB global workspace. If the variable does not exist, MATLAB Coder generates an error.

Defining Global Data in the MATLAB Global Workspace

To generate a MEX function for the `use_globals` function described in “Declare Global Variables” on page 19-81 using `codegen`:

- 1 In the MATLAB workspace, define and initialize the global data. At the MATLAB prompt, enter:

```
global AR B;  
AR = ones(4);  
B=[1 2 3];
```

- 2 Generate a MEX file.

```
codegen use_globals -args {0}  
% Use the -args option to specify that the input u  
% is a real, scalar, double  
% By default, codegen generates a MEX function,  
% use_globals_mex, in the current folder
```

Defining Global Data in a MATLAB Coder Project

- 1 On the project **Overview** tab, click **Add global** and enter a name for the global variable.

By default, MATLAB Coder names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, etc.

- 2 After adding a global variable, before building the project, specify its type and initial value. For more information, see “Specifying Global Variable Type and Initial Value in a Project” on page 16-33.

Note If you do not specify the type, you must create a variable with the same name in the global workspace.

Defining Global Data at the Command Line

To define global data at the command line, use the `codegen -globals` option. For example, to compile the `use_globals` function described in “Declare Global Variables” on page 19-81, specify two global inputs `AR` and `B` at the command line. Use the `-args` option to specify that the input `u` is a real, scalar double. By default, `codegen` generates a MEX function, `use_globals_mex`, in the current folder.

```
codegen -globals {'AR',ones(4),'B',[1 2 3]} use_globals -args {0}
```

Alternatively, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`.

Defining Variable-Size Global Data. To provide initial values for variable-size global data, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For example, to specify a global variable `g1` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
codegen foo -globals {'g1', {coder.typeof(0, [2 2],1),[1 1]}}
```

For a detailed explanation of the syntax, see `coder.typeof`.

Synchronizing Global Data with MATLAB

Why Synchronize Global Data?

The generated MEX function and MATLAB each have their own copies of global data. To make these copies consistent, you must synchronize their global data whenever the two interact. If you do not synchronize the data,

their global variables might differ. The level of interaction determines when to synchronize global data. For more information, see “When to Synchronize Global Data” on page 19-84.

When to Synchronize Global Data

By default, synchronization between the MEX function’s global data and MATLAB occurs at MEX function entry and exit and for extrinsic calls. Use this synchronization method for maximum consistency between the MEX function and MATLAB.

To improve performance, you can:

- Select to synchronize only at MEX function entry and exit points.
- Disable synchronization when the global data does not interact.
- Choose whether to synchronize before and after each extrinsic call.

The following table summarizes which global data synchronization options to use. To learn how to set these options, see “How to Synchronize Global Data” on page 19-85.

Global Data Synchronization Options

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Have maximum consistency when all extrinsic calls modify global data.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Default behavior.
Have maximum consistency when most extrinsic calls modify global data, but a few do not.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Use the <code>coder.extrinsic-sync:off</code> option to turn off synchronization for the extrinsic calls that do not change global data.
Have maximum consistency when most extrinsic calls do not modify global data, but a few do.	At MEX-function entry and exit	Yes. Use the <code>coder.extrinsic-sync:on</code> option to synchronize only the calls that modify global data.
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	At MEX-function entry and exit	No.
Communicate between generated MEX functions only. No interaction between MATLAB and MEX function global data.	Disabled	No.

How to Synchronize Global Data

To control global data synchronization, set the global data synchronization mode and select whether to synchronize extrinsic functions. For guidelines on which options to use, see “When to Synchronize Global Data” on page 19-84.

You can control the global data synchronization mode from the project settings dialog box, the command line, or a MEX configuration dialog box. You control the synchronization of data with extrinsic functions using the `coder.extrinsic -sync:on` and `-sync:off` options.

Controlling the Global Data Synchronization Mode in the Project Settings Dialog Box.

- 1 On the MATLAB Coder project **Build** tab, verify that **Output type** is set to MEX Function and then click the More settings link.
- 2 On the **Project Settings** dialog box **Memory** tab, set **Global data synchronization mode** to At MEX-function entry and exit or Disabled, as applicable.

Controlling the Global Data Synchronization Mode from the Command Line.

- 1 In the MATLAB workspace, define the code generation configuration object. At the MATLAB command line, enter:

```
mexcfg = coder.config('mex');
```

- 2 At the MATLAB command line, set the `GlobalDataSyncMethod` property to `SyncAtEntryAndExits` or `NoSync`, as applicable. For example:

```
mexcfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
```

- 3 When compiling your code, use the `mexcfg` configuration object. For example, to generate a MEX function for function `foo` that has no inputs:

```
codegen -config mexcfg foo
```

Controlling Synchronization for Extrinsic Function Calls. To control whether synchronization between MATLAB and MEX function global data occurs before and after you call an extrinsic function, use the `coder.extrinsic-sync:on` and `-sync:off` options.

By default, global data is:

- Synchronized before and after each extrinsic call, if the global data synchronization mode is At MEX-function entry, exit and extrinsic calls. If you are sure that certain extrinsic calls do not change global data, turn off synchronization for these calls using the `-sync:off` option. For example, if functions `foo1` and `foo2` do not change global data, turn off synchronization for these functions:

```
coder.extrinsic('-sync:off', 'foo1', 'foo2');
```

- Not synchronized, if the global data synchronization mode is At MEX-function entry and exit. If the code has a few extrinsic calls that change global data, turn on synchronization for these calls using the `-sync:on` option. For example, if functions `foo1` and `foo2` change global data, turn on synchronization for these functions:

```
coder.extrinsic('-sync:on', 'foo1', 'foo2');
```

- Not synchronized, if the global data synchronization mode is Disabled. When synchronization is disabled, you cannot use the `-sync:on` option to control the synchronization for specific extrinsic calls.

Limitations of Using Global Data

You cannot use global data with the `coder.cstructname` function.

Generation of Traceable Code

In this section...

“About Code Traceability” on page 19-88

“Generate Traceable Code” on page 19-89

“Format of Traceability Tags” on page 19-91

“Location of Comments in Generated Code” on page 19-91

“Traceability Limitations” on page 19-96

About Code Traceability

You can configure MATLAB Coder to generate C code and MEX functions that include the MATLAB source code as comments. Including this information in the generated code enables you to:

- Correlate the generated code with your source code.
- Understand how the generated code implements your algorithm.
- Evaluate the quality of the generated code.

In these automatically generated comments, a traceability tag immediately precedes each line of source code. This traceability tag provides details about the location of the source code. For more information, see “Format of Traceability Tags” on page 19-91.

For Embedded Coder projects, (requires an Embedded Coder license), you can also generate C/C++ code that includes the MATLAB function help text. The function help text is the first comment after the MATLAB function signature. It is displayed in the function banner of the generated code. The function help text provides information about the capabilities of the function and how to use it. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

Generate Traceable Code

To generate more traceable code, include MATLAB source code as comments in the generated code from the **Project Settings** dialog box, the command line, or a MEX configuration dialog box.

In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.

Note MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you switch the output type between MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

- 3 In the **Project Settings** dialog box, click the **Comments** tab.
- 4 On the **Code Appearance** tab, select **MATLAB source code as comments** and then close the dialog box.

At the Command Line

For MEX Targets. Use the `MATLABSourceComments` option of the MEX configuration object. For example, to compile the file `foo.m` and include the source code as comments in the generated MEX function:

- 1 In the MATLAB workspace, define the MEX configuration object by issuing a constructor command:

```
mexcfg = coder.config('mex');
```

- 2 From the command line, enable the `MATLABSourceComments`:

```
mexcfg.MATLABSourceComments = true;
```

- 3 Using the `-config` option, pass the configuration object to `codegen`. For example, to generate a MEX function for a function `foo` that has no input parameters:

```
codegen -config mexcfg foo
```

For C/C++ Libraries. Use the `MATLABSourceComments` option of the code generation configuration object. For example, to compile the file `foo.m` and include the source code as comments in the generated code for a C static library:

- 1 Create a code generation configuration object and enable the `MATLABSourceComments` option. For example, to create a configuration object for a static library:

```
cfg = coder.config('lib');  
% If an Embedded Coder license is available,  
% cfg is a coder.EmbeddedCodeConfig object,  
% otherwise it's a coder.CodeConfig object  
cfg.MATLABSourceComments = true;
```

- 2 Using the `-config` option, pass the configuration object to `codegen`. For example, to generate a library for a function `foo` that has no input parameters:

```
codegen -config cfg foo
```

For Embedded Coder projects (requires an Embedded Coder license), you can also include the function help text in the generated code function banner using the `MATLABFcnDesc` option. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

For C/C++ Executables. Use the `MATLABSourceComments` option of the code generation configuration object. For example, to compile the file `foo.m` and include the source code as comments in the generated code for a C executable:

- 1 Create a code generation configuration object and enable the `MATLABSourceComments` option. For example, to create a configuration object for a library:

```
cfg = coder.config('exe');
```

```
% If an Embedded Coder license is available,
% cfg is a coder.EmbeddedCodeConfig object,
% otherwise it's a coder.CodeConfig object
cfg.MATLABSourceComments = true;
```

- 2** Using the `-config` option, pass the configuration object to `codegen`. For example, to generate an executable for a function `foo` that has no input parameters:

```
codegen -config cfg main.c foo
% You must specify a main file when generating an executable
```

For Embedded Coder projects, (requires an Embedded Coder license), you can also include the function help text in the function banner of the generated code using the `MATLABFcnDesc` option. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

Format of Traceability Tags

In the generated code, traceability tags appear immediately before the MATLAB source code in the comment. The format of the tag is:
`<filename>:<line number>`.

For example, the comment indicates that the code `x = r * cos(theta);` appears at line 4 in the source file `straightline.m`.

```
/* 'straightline:4' x = r * cos(theta); */
```

Note With an Embedded Coder license, the traceability tags in the code generation report are hyperlinks to the MATLAB source code. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

Location of Comments in Generated Code

The auto-generated comments containing the source code and traceability tag appear in the generated code as follows.

Straight-Line Source Code

In straight-line source code without `if`, `while`, `for` or `switch` statements, the comment containing the source code precedes the generated code that implements the source code statement. This comment appears after user comments that precede the generated code.

For example, in the following code, the user comment, `/* Convert polar to Cartesian */`, appears before the automatically generated comment containing the first line of source code, together with its traceability tag, `/* 'straightline:4' x = r * cos(theta); */`.

MATLAB Code.

```
function [x y] = straightline(r,theta)
%#codegen
% Convert polar to Cartesian
x = r * cos(theta);
y = r * sin(theta);
```

Commented C Code.

```
void straightline(real_T r, real_T theta, ...
    real_T *x, real_T *y)
{
    /* Convert polar to Cartesian */
    /* 'straightline:4' x = r * cos(theta); */
    *x = r * muDoubleScalarCos(theta);
    /* 'straightline:5' y = r * sin(theta); */
    *y = r * muDoubleScalarSin(theta);
}
```

If Statements

The comment for the `if` statement immediately precedes the code that implements the statement. This comment appears after user comments that precede the generated code. The comments for the `elseif` and `else` clauses appear immediately after the code that implements the clause, and before the code generated for statements in the clause.

MATLAB Code.

```
function y = ifstmt(u,v)
%#codegen
if u > v
    y = v + 10;
elseif u == v
    y = u * 2;
else
    y = v - 10;
end
```

Commented C Code.

```
real_T ifstmt(real_T u, real_T v)
{
    /* 'ifstmt:3' if u > v */
    if(u > v) {
        /* 'ifstmt:4' y = v + 10; */
        return v + 10.0;
    } else if(u == v) {
        /* 'ifstmt:5' elseif u == v */
        /* 'ifstmt:6' y = u * 2; */
        return u * 2.0;
    } else {
        /* 'ifstmt:7' else */
        /* 'ifstmt:8' y = v - 10; */
        return v - 10.0;
    }
}
```

For Statements

The comment for the for statement header immediately precedes the generated code that implements the header. This comment appears after user comments that precede the generated code.

MATLAB Code.

```
function y = forstmt(u)
```

```
 %#codegen  
 y = 0;  
 for i=1:u  
     y = y + 1;  
 end
```

Commented C Code.

```
real_T forstmt(real_T u)  
{  
    real_T y;  
    real_T i;  
    /* 'forstmt:3' y = 0; */  
    y = 0.0;  
    /* 'forstmt:4' for i=1:u */  
    for(i = 1.0; i <= u; i++) {  
        /* 'forstmt:5' y = y + 1; */  
        y++;  
    }  
    return y;  
}
```

While Statements

The comment for the while statement header immediately precedes the generated code that implements the statement header. This comment appears after user comments that precede the generated code.

MATLAB Code.

```
function y = subfcn(y)  
coder.inline('never');  
while y < 100  
    y = y + 1;  
end
```

Commented C Code.

```
static void m_refp1_subfcn(real_T *y)  
{
```



```

/* 'whilestmt:6' coder.inline('never'); */
/* 'whilestmt:7' while y < 100 */
while(*y < 100.0) {
    /* 'whilestmt:8' y = y + 1; */
    (*y)++;
}
}

```

Switch Statements

The comment for the switch statement header immediately precedes the generated code that implements the statement header. This comment appears after user comments that precede the generated code. The comments for the case and otherwise clauses appear immediately after the generated code that implements the clause, and before the code generated for statements in the clause.

MATLAB Code.

```

function y = switchstmt(u)
%#codegen
y = 0;
switch u
    case 1
        y = y + 1;
    case 3
        y = y + 2;
    otherwise
        y = y - 1;
end

```

Commented C Code.

```

real_T switchstmt(real_T u)
{
    /* 'switchstmt:3' y = 0; */
    /* 'switchstmt:4' switch u */
    switch((int32_T)u) {
    case 1:

```

```
        /* 'switchstmt:5' case 1 */
        /* 'switchstmt:6' y = y + 1; */
        return 1.0;
        break;
    case 3:
        /* 'switchstmt:7' case 3 */
        /* 'switchstmt:8' y = y + 2; */
        return 2.0;
        break;
    default:
        /* 'switchstmt:9' otherwise */
        /* 'switchstmt:10' y = y - 1; */
        return -1.0;
        break;
    }
}
```

Traceability Limitations

For MATLAB Coder, there are traceability limitations:

- You cannot include MATLAB source code as comments for:
 - MathWorks toolbox functions
 - P-code
- The appearance or location of comments can vary depending on the following conditions:
 - Even if the implementation code is eliminated, for example, due to constant folding, comments might still appear in the generated code.
 - If a complete function or code block is eliminated, comments might be eliminated from the generated code.
 - For certain optimizations, the comments might be separated from the generated code.
 - Even if you do not choose to include source code comments in the generated code, the generated code includes legally required comments from the MATLAB source code.

Generate Code for Enumerated Types

When generating MEX functions from MATLAB code, use enumerated types based on `int32` with MATLAB Coder . When generating C code with MATLAB Coder, you can also use this enumerated type, but `int32` does not provide methods for customizing the behavior of enumerated data.

Generate Code for Variable-Size Data

In this section...

“Disable Support for Variable-Size Data” on page 19-98

“Control Dynamic Memory Allocation” on page 19-99

“Generating Code for MATLAB Functions with Variable-Size Data” on page 19-101

“Generate Code for a MATLAB Function That Expands a Vector in a Loop” on page 19-103

“Using Dynamic Memory Allocation for an "Atoms" Simulation” on page 19-110

Variable-size data is data whose size might change at run time. You can use MATLAB Coder to generate C/C++ code from MATLAB code that uses variable-size data. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. By default, for MEX and C/C++ code generation, support for variable-size data is enabled and dynamic memory allocation is enabled for variable-size arrays whose size exceeds a configurable threshold.

Disable Support for Variable-Size Data

By default, for MEX and C/C++ code generation, support for variable-size data is enabled. You modify variable sizing settings from the project settings dialog box, the command line, or using dialog boxes.

In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More** settings link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **General** tab.

- 4 On the **Memory** tab, select or clear **Enable variable-sizing**. Close the dialog box.

At the Command Line

- 1 Create a configuration object for code generation. For example, for a library:

```
cfg = coder.config('lib');
```

- 2 Set the `EnableVariableSizing` option:

```
cfg.EnableVariableSizing = false;
```

- 3 Using the `-config` option, pass the configuration object to `codegen` :

```
codegen -config cfg foo
```

Control Dynamic Memory Allocation

By default, dynamic memory allocation is enabled for variable-size arrays whose size exceeds a configurable threshold. If you disable support for variable-size data (see “Disable Support for Variable-Size Data” on page 19-98), you also disable dynamic memory allocation. You can modify dynamic memory allocation settings from the project settings dialog box or the command line.

In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **Memory** tab.
- 4 On the **Memory** tab, set **Dynamic memory allocation** to one of the following options:

Setting	Action
Never	Dynamic memory allocation is disabled. Variable-size data is allocated statically on the stack.
For all variable-sized arrays	Dynamic memory allocation is enabled for variable-size arrays. Variable-size data is allocated dynamically on the heap.
For arrays with maximum size above threshold	Dynamic memory allocation is enabled for variable-size arrays whose size exceeds the Dynamic memory allocation threshold . Variable-size arrays whose size is less than this threshold are allocated on the stack.

5 Optionally, if you set **Dynamic memory allocation** to For arrays with maximum size above threshold, configure **Dynamic memory allocation threshold** to fine tune memory allocation.

6 Close the dialog box.

At the Command Line

1 Create a configuration object for code generation. For example, for a MEX function:

```
mexcfg = coder.config('mex');
```

2 Set the DynamicMemoryAllocation option:

Setting	Action
<code>mexcfg.DynamicMemoryAllocation='Off';</code>	Dynamic memory allocation is disabled. Variable-size data is allocated statically on the stack.
<code>mexcfg.DynamicMemoryAllocation='AllVariableSizeArrays';</code>	Dynamic memory allocation is enabled for variable-size arrays. Variable-size data is allocated dynamically on the heap.
<code>mexcfg.DynamicMemoryAllocation='Threshold';</code>	Dynamic memory allocation is enabled for variable-size arrays whose size (in bytes) is greater than or equal to the value specified using the Dynamic memory allocation threshold parameter. Variable-size arrays whose size is less than this threshold are allocated on the stack.

3 Optionally, if you set Dynamic memory allocation to `'Threshold'`, configure Dynamic memory allocation threshold to fine tune memory allocation.

4 Using the `-config` option, pass the configuration object to `codegen`:

```
codegen -config mexcfg foo
```

Generating Code for MATLAB Functions with Variable-Size Data

Here is a basic workflow that first generates MEX code for verifying the generated code and then generates standalone code after you are satisfied with the result of the prototype.

To work through these steps with a simple example, see “Generate Code for a MATLAB Function That Expands a Vector in a Loop” on page 19-103

- 1** In the MATLAB Editor, add the compilation directive `codegen` at the top of your function.

This directive:

- Indicates that you intend to generate code for the MATLAB algorithm
- Turns on checking in the MATLAB Code Analyzer to detect potential errors during code generation

- 2** Address issues detected by the Code Analyzer.

In some cases, the MATLAB Code Analyzer warns you when your code assigns data a fixed size but later grows the data, such as by assignment or concatenation in a loop. If that data is supposed to vary in size at run time, you can ignore these warnings.

- 3** Generate a MEX function using `codegen` to verify the generated code. Use the following command-line options:

- `-args {coder.typeof...}` if you have variable-size inputs
- `-report` to generate a code generation report

For example:

```
codegen -report foo -args {coder.typeof(0,[2 4],1)}
```

This command uses `coder.typeof` to specify one variable-size input for function `foo`. The first argument, `0`, indicates the input data type (`double`) and complexity (`real`). The second argument, `[2 4]`, indicates the size, a matrix with two dimensions. The third argument, `1`, indicates that the input is variable sized. The upper bound is 2 for the first dimension and 4 for the second dimension.

Note During compilation, codegen detects variables and structure fields that change size after you define them, and reports these occurrences as errors. In addition, codegen performs a run-time check to generate errors when data exceeds upper bounds.

4 Fix size mismatch errors:

Cause:	How To Fix:	For More Information:
You try to change the size of data after its size has been locked.	Declare the data to be variable sized.	See “Diagnosing and Fixing Size Mismatch Errors” on page 7-23.

5 Fix upper bounds errors

Cause:	How To Fix:	For More Information:
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	See “Specifying Upper Bounds for Variable-Size Data” on page 7-6 and “Diagnosing and Fixing Size Mismatch Errors” on page 7-23.
MATLAB attempts to compute an upper bound for unbounded variable-size data.	If the data is unbounded, enable dynamic memory allocation.	See “Control Dynamic Memory Allocation” on page 19-99.

6 Generate C/C++ code using the codegen function.

Generate Code for a MATLAB Function That Expands a Vector in a Loop

- “About the MATLAB Function uniquetol” on page 19-104

- “Step 1: Add Compilation Directive for Code Generation” on page 19-104
- “Step 2: Address Issues Detected by the Code Analyzer” on page 19-105
- “Step 3: Generate MEX Code” on page 19-105
- “Step 4: Fix the Size Mismatch Error” on page 19-107
- “Step 5: Generate C Code” on page 19-108
- “Step 6: Change the Dynamic Memory Allocation Threshold” on page 19-109

About the MATLAB Function `uniquetol`

This example uses the function `uniquetol`. This function returns in vector `B` a version of input vector `A`, where the elements are unique to within tolerance `tol` of each other. In vector `B`, $\text{abs}(B(i) - B(j)) > \text{tol}$ for all `i` and `j`. Initially, assume input vector `A` can store up to 100 elements.

```
function B = uniquetol(A, tol)
A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

Step 1: Add Compilation Directive for Code Generation

Add the `%codegen` compilation directive at the top of the function:

```
function B = uniquetol(A, tol) %codegen
A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

```

    end
end

```

Step 2: Address Issues Detected by the Code Analyzer

The Code Analyzer detects that variable B might change size in the for-loop. It issues this warning:

The variable 'B' appears to change size on every loop iteration. Consider preallocating for speed.

In this function, vector B should expand in size as it adds values from vector A. Therefore, you can ignore this warning.

Step 3: Generate MEX Code

To generate MEX code, use the `codegen` function.

- 1 Generate a MEX function for `uniquetol`:

```
codegen -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

What do these command-line options mean?

The `-args` option specifies the class, complexity, and size of each input to function `uniquetol`:

- The first argument, `coder.typeof`, defines a variable-size input. The expression `coder.typeof(0,[1 100],1)` defines input A as a real double vector with a fixed upper bound. Its first dimension is fixed at 1 and its second dimension can vary in size up to 100 elements.

For more information, see “Specify Variable-Size Inputs at the Command Line” on page 19-48.

- The second argument, `coder.typeof(0)`, defines input `tol` as a real double scalar.

The `-report` option instructs `codegen` to generate a code generation report, regardless of whether errors or warnings occur.

For more information, see the codegen reference page.

Executing this command generates a compiler error:

```
??? Size mismatch (size [1 x 1] ~= size [1 x 2]).
The size to the left is the size
of the left-hand side of the assignment.
```

2 Open the error report and select the **Variables** tab.

```
1 function B = uniquetol(A, tol) %#codegen
2 A = sort(A);
3 B = A(1);
4 k = 1;
5 for i = 2:length(A)
6     if abs(A(k) - A(i)) > tol
7         B = [B A(i)];
8         k = i;
9     end
10 end
```

Summary	All Messages (1)	Variables			
Order	Variable	Type	Size	Complex	Class
1	B	Output	1 x 1	No	double
2	A > 1	Input	1 x :100	No	double
3	A > 2	Local	1 x :?	No	double
4	tol	Input	1 x 1	No	double
5	k	Local	1 x 1	No	double
6	i	Local	1 x 1	No	double

The error indicates a size mismatch between the left-hand side and right-hand side of the assignment statement `B = [B A(i)];`. The assignment `B = A(1)` establishes the size of `B` as a fixed-size scalar (1 x 1). Therefore, the concatenation of `[B A(i)]` creates a 1 x 2 vector.

Step 4: Fix the Size Mismatch Error

To fix this error, declare B to be a variable-size vector.

- 1 Add this statement to the `uniquetol` function:

```
coder.varsize('B');
```

It should appear before B is used (read). For example:

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);
```

```
coder.varsize('B');
```

```
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

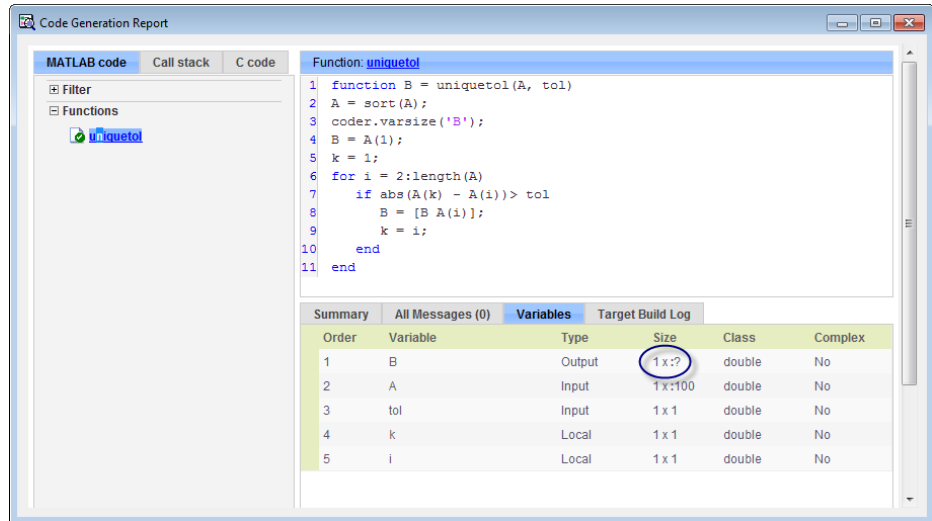
The function `coder.varsize` declares every instance of B in `uniquetol` to be variable sized.

- 2 Generate code again using the same command:

```
codegen -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

In the current folder, `codegen` generates a MEX function for `uniquetol` and provides a link to the code generation report.

- 3 Click the *View report* link.
- 4 In the code generation report, select the **Variables** tab.



The size of variable B is 1x?, indicating that it is variable size with no upper bounds.

Step 5: Generate C Code

Generate C code for variable-size inputs. By default, `codegen` allocates memory statically for data whose size is less than the dynamic memory allocation threshold of 64 kilobytes. If the size of the data exceeds the threshold or is unbounded, `codegen` allocates memory dynamically on the heap.

- 1 Create a configuration option for C library generation:

```
cfg=coder.config('lib');
```

- 2 Issue this command:

```
codegen -config cfg -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

`codegen` generates a static library in the default location, `codegen\lib\uniquetol` and provides a link to the code generation report.

- 3 Click the *View report* link.
- 4 In the code generation report, click the **C code** tab.
- 5 On the **C code** tab, click the link to `uniquetol.h`.

The function declaration is:

```
extern void uniquetol(const real_T A_data[100], const int32_T A_size[2],...
    real_T tol, emxArray_real_T *B);
```

codegen computes the size of A and, because its maximum size is less than the default dynamic memory allocation threshold of 64k bytes, allocates this memory statically. The generated code contains two pieces of information about A:

- `real_T A_data[100]`: the maximum size of input A (where 100 is the maximum size specified using `coder.typeof`).
- `int32_T A_sizes[2]`: the actual size of the input.

Because B is variable size with unknown upper bounds, in the generated code, codegen represents B as `emxArray_real_T`. MATLAB provides utility functions for creating and interacting with `emxArrays` in your generated code. For more information, see “C Code Interface for Arrays” on page 7-19.

Step 6: Change the Dynamic Memory Allocation Threshold

In this step, you reduce the dynamic memory allocation threshold and generate code for an input that exceeds this threshold.

- 1 Set the dynamic memory allocation threshold to 4 kilobytes and generate code where the size of input A exceeds this threshold.

```
cfg.DynamicMemoryAllocationThreshold=4096;
codegen -config cfg -report uniquetol -args {coder.typeof(0,[1 10000],1),coder.typeof(0)}
```

- 2 View the generated code in the report. Because the maximum size of input A now exceeds the dynamic memory allocation threshold, codegen allocates A dynamically on the heap and represents A as `emxArray_real_T`.

```
extern void uniquetol(const mxArray_real_T *A, ...
    real_T tol, mxArray_real_T *B);
```

Using Dynamic Memory Allocation for an "Atoms" Simulation

This example shows how to generate code for a MATLAB algorithm that runs a simulation of bouncing "atoms" and returns the result after a number of iterations. There are no upper bounds on the number of atoms that the algorithm accepts, so this example takes advantage of dynamic memory allocation.

Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will contain only the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_atoms');
```

About the 'run_atoms' Function

The `run_atoms.m` function runs a simulation of bouncing atoms (also applying gravity and energy loss).

```
help run_atoms
```

```
atoms = run_atoms(atoms,n)
atoms = run_atoms(atoms,n,iter)
Where 'atoms' the initial and final state of atoms (can be empty)
'n' is the number of atoms to simulate.
'iter' is the number of iterations for the simulation
(if omitted it is defaulted to 3000 iterations.)
```

Set Up Code Generation Options

Create a code generation configuration object


```
cfg = coder.config;
% Enable dynamic memory allocation for variable size matrices.
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

Set Up Example Inputs

Create a template structure 'Atom' to provide the compiler with the necessary information about input parameter types. An atom is a structure with four fields (x,y,vx,vy) specifying position and velocity in Cartesian coordinates.

```
atom = struct('x', 0, 'y', 0, 'vx', 0, 'vy', 0);
```

Generate a MEX Function for Testing

Use the command 'codegen' with the following arguments:

'-args {coder.typeof(atom, [1 Inf]),0,0}' indicates that the first argument is a row vector of atoms where the number of columns is potentially infinite. The second and third arguments are scalar double values.

'-config cfg' enables dynamic memory allocation, defined by workspace variable cfg

```
codegen run_atoms -args {coder.typeof(atom, [1 Inf]),0,0} -config cfg -o ru
```

Run the MEX Function

The MEX function simulates 10000 atoms in approximately 1000 iteration steps given an empty list of atoms. The return value is the state of all the atoms after simulation is complete.

```
atoms = run_atoms_mex([],10000,1000)
```

```
atoms =
```

```
1x10000 struct array with fields:
```

```
  x
  y
```

```
vx  
vy
```

Run the MEX Function Again

Continue the simulation with another 500 iteration steps

```
atoms = run_atoms_mex(atoms,10000,500)
```

```
atoms =
```

```
1x10000 struct array with fields:
```

```
x  
y  
vx  
vy
```

Generate a Standalone C Code Library

To generate a C library, create a standard configuration object for libraries:

```
cfg = coder.config('lib');
```

Enable dynamic memory allocation

```
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

In MATLAB the default data type is double. However, integers are usually used in C code, so pass int32 integer example values to represent the number of atoms and iterations.

```
codegen run_atoms -args {coder.typeof(atom, [1 Inf]),int32(0),int32(0)} -co
```

Inspect Generated Code

When creating a library the code is generated in the folder `codegen/lib/run_atoms/`. The code in this folder is self contained. To interface

with the compiled C code you need only the generated header files and the library file.

```
dir codegen/lib/run_atoms
```

```
.          rtw_proj.tmw          run_atoms_initialize.h
..         rtwtypes.h        run_atoms_initialize.ob
buildInfo.mat  run_atoms.c    run_atoms_ref.rsp
codeInfo.mat  run_atoms.h    run_atoms_rtw.bat
rtGetInf.c    run_atoms.lib   run_atoms_rtw.mk
rtGetInf.h    run_atoms.obj   run_atoms_rtw_tools.mk
rtGetInf.obj  run_atoms_emxAPI.c  run_atoms_terminate.c
rtGetNaN.c    run_atoms_emxAPI.h  run_atoms_terminate.h
rtGetNaN.h    run_atoms_emxAPI.obj run_atoms_terminate.obj
rtGetNaN.obj  run_atoms_emxutil.c run_atoms_types.h
rt_nonfinite.c  run_atoms_emxutil.h
rt_nonfinite.h  run_atoms_emxutil.obj
rt_nonfinite.obj run_atoms_initialize.c
```

Write a C Main Function

Typically, the main function is platform-dependent code that performs rendering or some other processing. In this example, a pure ANSI-C function produces a file 'run_atoms_state.m' which (when run) contains the final state of the atom simulation.

```
type run_atoms_main.c
```

```
/* Include standard C libraries */
#include <stdio.h>

/* The interface to the main function we compiled. */
#include "codegen/lib/run_atoms/run_atoms.h"

/* The interface to EMX data structures. */
#include "codegen/lib/run_atoms/run_atoms_emxAPI.h"
```

```
int main(int argc, char **argv)
{
    int i;
    emxArray_Atom *atoms;

    /* Main arguments unused */
    (void) argc;
    (void) argv;

    /* Initially create an empty row vector of atoms (1 row, 0 columns) */
    atoms = emxCreate_Atom(1, 0);

    /* Call the function to simulate 10000 atoms in 1000 iteration steps */
    run_atoms(atoms, 10000, 1000);

    /* Call the function again to do another 500 iteration steps */
    run_atoms(atoms, 10000, 500);

    /* Print the result to standard output */
    for (i = 0; i < atoms->size[1]; i++) {
        printf("%f %f %f %f\n",
            atoms->data[i].x, atoms->data[i].y, atoms->data[i].vx, atoms->d
    }

    /* Free memory */
    emxDestroyArray_Atom(atoms);
    return(0);
}
```

Create a Configuration Object for Executables

```
cfg = coder.config('exe');
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

Generate a Standalone Executable

You must pass the function (run_atoms.m) as well as custom C code (run_atoms_main.c) The 'codegen' command automatically generates C code

from the MATLAB code, then calls the C compiler to bundle this generated code with the custom C code (`run_atoms_main.c`).

```
codegen run_atoms run_atoms_main.c -args {coder.typeof(atom, [1 Inf]),int32}
```

Run the Executable

After simulation is complete, this produces the file `'atoms_state.mat'`. The MAT file is a 10000x4 matrix, where each row is the position and velocity of an atom (`x`, `y`, `vx`, `vy`) representing the current state of the whole system.

```
[~,atoms_data] = system(['.' filesep 'run_atoms']);  
fh = fopen('atoms_state.mat', 'w');  
fprintf(fh, '%s', atoms_data);  
fclose(fh);
```

Fetch the State

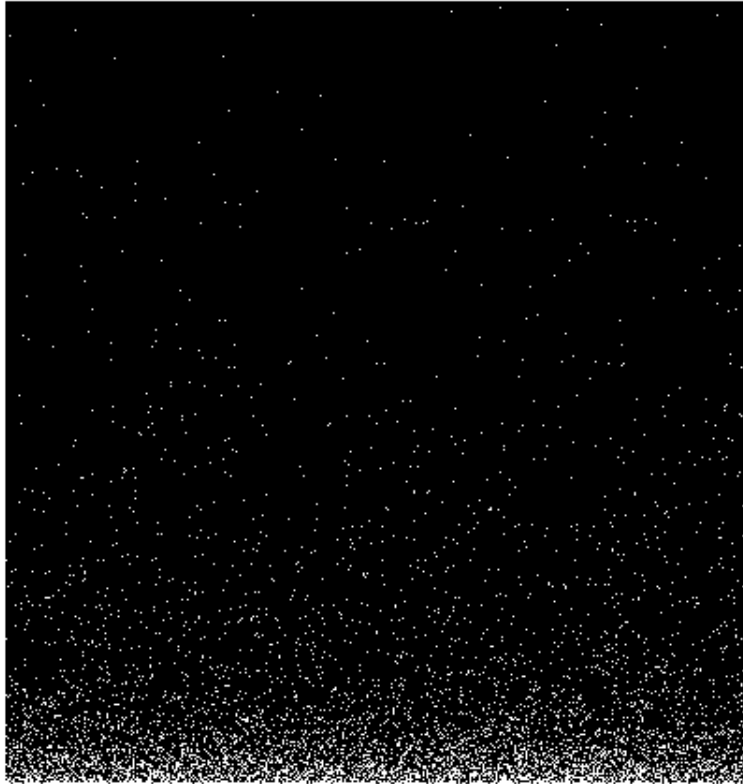
Running the executable produced `'atoms_state.mat'`. Now, recreate the structure array from the saved matrix

```
load atoms_state.mat -ascii  
clear atoms  
for i = 1:size(atoms_state,1)  
    atoms(1,i).x = atoms_state(i,1);  
    atoms(1,i).y = atoms_state(i,2);  
    atoms(1,i).vx = atoms_state(i,3);  
    atoms(1,i).vy = atoms_state(i,4);  
end
```

Render the State

Call `'run_atoms_mex'` with zero iterations to render only

```
run_atoms_mex(atoms, 10000, 0);
```



Clean Up

Remove files and return to original folder

Run Command: Cleanup

```
if ispc
    delete run_atoms.exe
else
```

```
        delete run_atoms
    end
    delete atoms_state.mat
cleanup
```

Code Generation for MATLAB Classes

You can generate code for MATLAB value and handle classes and user-defined System objects that inherit from a handle class. For more information, see “MATLAB Classes”.

How MATLAB Coder Partitions Generated Code

In this section...

“Partitioning Generated Files” on page 19-119

“How to Select the File Partitioning Method” on page 19-119

“Partitioning Generated Files with One C/C++ File Per MATLAB File” on page 19-120

“Generated Files and Locations” on page 19-125

“File Partitioning and Inlining” on page 19-128

Partitioning Generated Files

By default, during code generation, MATLAB Coder partitions the code to match your MATLAB file structure. This one-to-one mapping lets you easily correlate your files generated in C/C++ with the compiled MATLAB code. MATLAB Coder cannot produce the same one-to-one correspondence for MATLAB functions that are inlined in generated code (see “File Partitioning and Inlining” on page 19-128).

Alternatively, you can select to generate all C/C++ functions into a single file. For more information, see “How to Select the File Partitioning Method” on page 19-119. This option facilitates integrating your code with existing embedded software.

How to Select the File Partitioning Method

In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **Code Appearance** tab.

- 4 On the **Code Appearance** tab, set the **Generated file partitioning method** to **Generate one file for each MATLAB file** or **Generate all functions into a single file**. Close the dialog box.

At the Command Line

Use the codegen configuration object `FilePartitionMethod` option. For example, to compile the function `foo` that has no inputs and generate one C/C++ file for each MATLAB function:

- 1 Create a MEX configuration object and set the `FilePartitionMethod` option:

```
mexcfg = coder.config('mex');  
mexcfg.FilePartitionMethod = 'MapMFileToCFile';
```

- 2 Using the `-config` option, pass the configuration object to `codegen`:

```
codegen -config mexcfg -O disable:inline foo  
% Disable inlining to generate one C/C++ file for each MATLAB function
```

Partitioning Generated Files with One C/C++ File Per MATLAB File

By default, for MATLAB functions that are not inlined, MATLAB Coder generates one C/C++ file for each MATLAB file. In this case, MATLAB Coder partitions generated C/C++ code so that it corresponds to your MATLAB files.

How MATLAB Coder Partitions Entry-Point MATLAB Functions

For each entry-point (top-level) MATLAB function, MATLAB Coder generates one C/C++ source, header, and object file with the same name as the MATLAB file.

For example, suppose you define a simple function `foo` that calls the function `identity`. The source file `foo.m` contains the following code:

```
function y = foo(u,v) %#codegen  
s = single(u);  
d = double(v);  
y = double(identity(s)) + identity(d);
```

Here is the code for `identity.m` :

```
function y = identity(u) %#codegen
y = u;
```

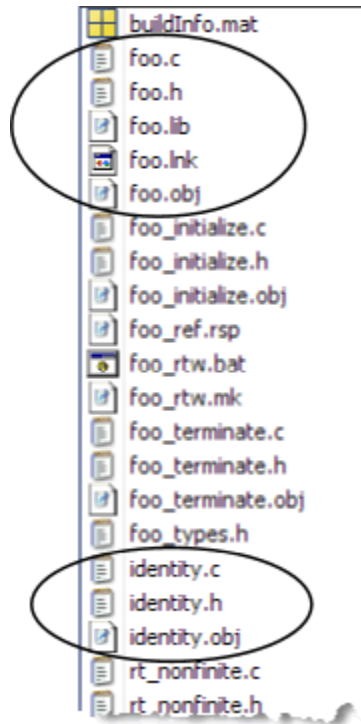
In the MATLAB Coder project interface, to generate a C static library for `foo.m`:

- 1** First, define the inputs `u` and `v`. For more information, see “Specifying Properties of Primary Function Inputs in a Project” on page 16-7.
- 2** In the MATLAB Coder project, click the **Build** tab.
- 3** On the **Build** tab:
 - a** Set the **Output type** to **C/C++ Static Library**.
 - b** Click the **More settings** link to view the project settings for the selected output type.
 - c** In the **Project Settings** dialog box, click the **All Settings** tab.
 - d** On this tab, under **Function Inlining**, set the **Inline threshold** parameter to **0**.
- 4** Click **Build** to generate a library.

To generate a C static library for `foo.m` at the command line, enter:

```
codegen -config:lib -O disable:inline foo -args {0, 0}
% Use the -args option to specify that u and v are both
% real, scalar doubles
```

MATLAB Coder generates source, header, and object files for `foo` and `identity` in your output folder.



How MATLAB Coder Partitions Local Functions

For each local function, MATLAB Coder generates code in the same C/C++ file as the calling function. For example, suppose you define a function `foo` that calls a local function `identity`:

```
function y = foo(u,v) %#codegen
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);

function y = identity(u)
y = u;
```

To generate a C++ library, before generating code, select a C++ compiler and set C++ as your target language. For example, at the command line:

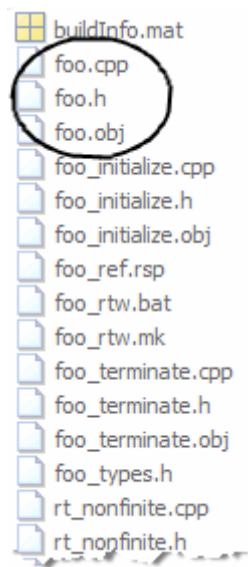
1 Select C++ as your target language:

```
cfg = coder.config('lib')
cfg.TargetLang='C++'
```

2 Generate the C++ library:

```
codegen -config cfg foo -args {0, 0}
% Use the -args option to specify that u and v are both
% real, scalar doubles
```

In the primary function `foo`, MATLAB Coder inlines the code for the identity local function.



Note If you specify C++, MATLAB Coder wraps the C code into `.cpp` files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Here is an excerpt of the generated code in `foo.cpp`:

```
...
/* Function Definitions */
real_T foo(real_T u, real_T v)
{
    return (real_T)(real32_T)u + v;
}
...
```

How MATLAB Coder Partitions Overloaded Functions

An overloaded function is a function that has multiple implementations to accommodate different classes of input. For each implementation (that is not inlined), MATLAB Coder generates a separate C/C++ file with a unique numeric suffix.

For example, suppose you define a simple function `multiply_defined`:

```
codegen
function y = multiply_defined(u)

y = u+1;
```

You then add two more implementations of `multiply_defined`, one to handle inputs of type `single` (in an `@single` subfolder) and another for inputs of type `double` (in an `@double` subfolder).

To call each implementation, define the function `call_multiply_defined`:

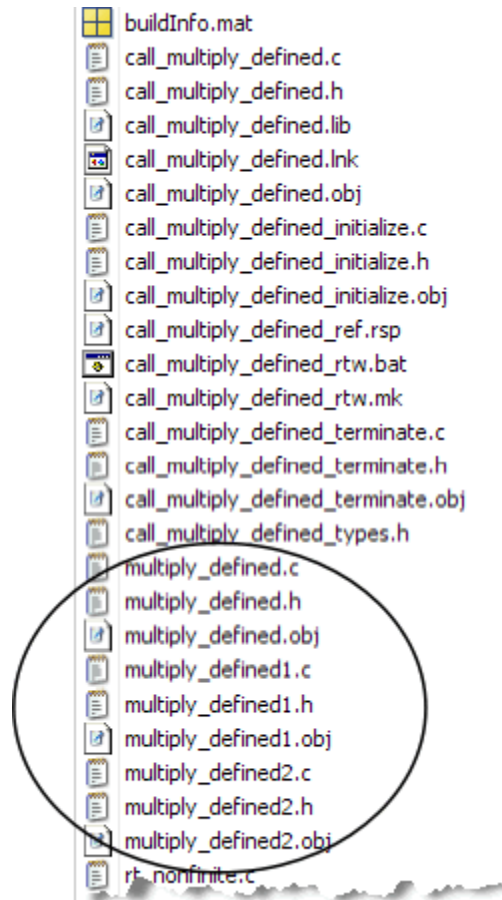
```
codegen
function [y1,y2,y3] = call_multiply_defined

y1 = multiply_defined(int32(2));
y2 = multiply_defined(2);
y3 = multiply_defined(single(2));
```

Next, generate C code for the overloaded function `multiply_defined`. For example, at the MATLAB command line, enter:

```
codegen -O disable:inline -config:lib call_multiply_defined
```

MATLAB Coder generates C source, header, and object files for each implementation of `multiply_defined`, as highlighted. Use numeric suffixes to create unique file names.



Generated Files and Locations

The types and locations of generated files depend on the target that you specify. For all targets, if errors or warnings occur during build or if you explicitly request a report, MATLAB Coder generates reports.

Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

Generated Files for MEX Targets

By default, MATLAB Coder generates the following files for MEX function (mex) targets.

Type of Files	Location
Platform-specific MEX files	Current folder
MEX, and C/C++ source, header, and object files	<code>codegen/mex/function_name</code>
HTML reports	<code>codegen/mex/function_name/html</code>

Generated Files for C/C++ Static Library Targets

By default, MATLAB Coder generates the following files for C/C++ static library targets.

Type of Files	Location
C/C++ source, library, header, and object files	<code>codegen/lib/function_name</code>
HTML reports	<code>codegen/lib/function_name/html</code>

Generated Files for C/C++ Dynamic Library Targets

By default, MATLAB Coder generates the following files for C/C++ dynamic library targets.

Type of Files	Location
C/C++ source, library, header, and object files	<code>codegen/dll/function_name</code>
HTML reports	<code>codegen/dll/function_name/html</code>

Generated Files for C/C++ Executable Targets

By default, MATLAB Coder generates the following files for C/C++ executable targets.

Type of Files	Location
C/C++ source, header, and object files	codegen/exe/ <i>function_name</i>
HTML reports	codegen/exe/ <i>function_name</i> /html

Changing Names and Locations of Generated Files

In the Project Settings Dialog Box.

To change the...	Do this...
Output file name	On the Build tab, enter the name in the Output file name field.
Output file location	<p>On the Build tab:</p> <ol style="list-style-type: none"> 1 Click the More settings link. 2 In the Project Settings dialog box, click the Paths tab. 3 On this tab, set Build folder to Specified folder. The Build folder name field appears. 4 For this field, either browse to the output file location or enter the full path. The path must not contain spaces.

To change the...	Do this...
	<p>Note The output file location should not contain:</p> <ul style="list-style-type: none"> • Spaces, as this can lead to code generation failures in certain operating system configurations. • Non 7-bit ASCII characters, such as Japanese characters.

At the Command Line. You can change the name and location of generated files by using the codegen options `-o` and `-d`.

File Partitioning and Inlining

How MATLAB Coder partitions generated C/C++ code depends on whether you choose to generate one C/C++ file for each MATLAB file and whether you inline your MATLAB functions.

If you...	MATLAB Coder...
Generate all C/C++ functions into a single file and disable inlining	Generates a single C/C++ file without inlining functions.
Generate all C/C++ functions into a single file and enable inlining	Generates a single C/C++ file. Inlines functions whose sizes fall within the inlining threshold.

If you...	MATLAB Coder...
Generate one C/C++ file for each MATLAB file and disable inlining	Partitions generated C/C++ code to match MATLAB file structure. See “Partitioning Generated Files with One C/C++ File Per MATLAB File” on page 19-120.
Generate one C/C++ file for each MATLAB file and enable inlining	Places inlined functions in the same C/C++ file as the function into which they are inlined. Even when you enable inlining, MATLAB Coder inlines only those functions whose sizes fall within the inlining threshold. For MATLAB functions that are not inlined, MATLAB Coder partitions the generated C/C++ code, as described.

Tradeoffs Between File Partitioning and Inlining

Weighing file partitioning against inlining represents a trade-off between readability, efficiency, and ease of integrating your MATLAB code with existing embedded software.

If You Generate...	Generated C/C++ Code	Advantages	Disadvantages
All C/C++ functions into a single file	Does not match MATLAB file structure	Easier to integrate with existing embedded software	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and enable inlining	Does not exactly match MATLAB file structure	Program executes faster	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and disable inlining	Matches MATLAB file structure	Easy to map C/C++ code to original MATLAB file	Program runs less efficiently

How Disabling Inlining Affects File Partitioning

Inlining is enabled by default. Therefore, to generate one C/C++ file for each top-level MATLAB function, you must:

- Select to generate one C/C++ file for each top-level MATLAB function. For more information, see “How to Select the File Partitioning Method” on page 19-119.
- Explicitly disable inlining, either globally or for individual MATLAB functions.

How to Disable Inlining Globally in the Project Settings Dialog Box.

- 1 In the MATLAB Coder project, click the **Build** tab.

- 2 On this tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **All Settings** tab.
- 4 On this tab, under **Function Inlining** set the **Inlining threshold** to zero. Close the dialog box.

How to Disable Inlining Globally at the Command Line. To disable inlining of functions, use the `-O disable:inline` option with `codegen`. For example, to disable inlining and generate a MEX function for a function `foo` that has no inputs:

```
codegen -O disable:inline foo
```

For more information, see the description of `codegen`.

How to Disable Inlining for Individual Functions. To disable inlining for an individual MATLAB function, add the directive `coder.inline('never')`; on a separate line in the source MATLAB file, after the function signature.

```
function y = foo(u,v) %#codegen
coder.inline('never');
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

`codegen` does not inline entry-point functions.

The `coder.inline` directive applies only to the function in which it appears. In this example, inlining is disabled for function `foo`, but not for `identity`, a top-level function defined in a separate MATLAB file and called by `foo`. To disable inlining for `identity`, add this directive after its function signature in the source file `identity.m`. For more information, see `coder.inline`.

For a more efficient way to disable inlining for both functions, see “How to Disable Inlining Globally at the Command Line” on page 19-131.

Correlating C/C++ Code with Inlined Functions

To correlate the C/C++ code that you generate with the original inlined functions, add comments in the MATLAB code to identify the function. These comments will appear in the C/C++ code and help you map the generated code back to the original MATLAB functions.

Modifying the Inlining Threshold

To change inlining behavior, adjust the inlining threshold parameter.

Modifying the Inlining Threshold in the Project Settings Dialog Box.

On the **Project Settings** dialog box **All Settings** tab, under **Function Inlining**, set the value of the **Inline threshold** parameter.

Modifying the Inlining Threshold at the Command Line. Set the value of the `InlineThreshold` parameter of the configuration object. See `coder.MexCodeConfig`, `coder.CodeConfig`, `coder.EmbeddedCodeConfig`.

Customize the Post-Code-Generation Build Process

In this section...

“Workflow for Customizing Post-Code-Generation Builds” on page 19-133

“Build Information Object” on page 19-133

“Build Information Functions” on page 19-134

“Programming a Post-Code-Generation Command” on page 19-172

“Using a Post-Code-Generation Command in Your Build” on page 19-172

“Programming and Using a Post-Code-Generation Command at the Command Line” on page 19-174

Workflow for Customizing Post-Code-Generation Builds

For certain applications, you might want to control aspects of the build process that occur after code generation but before compilation. For example, you might want to specify compiler or linker options. You can customize build processing that occurs after code generation using MATLAB Coder for MEX functions, C/C++ libraries and C/C++ executables.

To customize a post-code-generation build:

- 1 Program a post-code-generation command. Typically, you use this command to get the project name and build information or to add data to the build information object.
- 2 Use this command in your build.

Build Information Object

At the start of a build, the MATLAB Coder build process logs the following project, build option, and dependency information to a temporary build information object, `RTW.BuildInfo`:

- Compiler options
- Preprocessor identifier definitions

- Linker options
- Source files and paths
- Include files and paths
- Precompiled external libraries

Use the “Build Information Functions” on page 19-134 to access this information in the build information object. “Programming a Post-Code-Generation Command” on page 19-172 explains how to use the functions to control a post-code-generation build.

When code generation is complete, MATLAB Coder creates a `buildInfo.mat` file in the build folder.

Build Information Functions

- “addCompileFlags” on page 19-135
- “addDefines” on page 19-136
- “addIncludeFiles” on page 19-138
- “addIncludePaths” on page 19-140
- “addLinkFlags” on page 19-142
- “addLinkObjects” on page 19-143
- “addNonBuildFiles” on page 19-147
- “addSourceFiles” on page 19-149
- “addSourcePaths” on page 19-151
- “addTMFTokens” on page 19-154
- “findIncludeFiles” on page 19-156
- “getCompileFlags” on page 19-157
- “getDefines” on page 19-157
- “getFullFileList” on page 19-159
- “getIncludeFiles” on page 19-160

- “getIncludePaths” on page 19-161
- “getLinkFlags” on page 19-162
- “getNonBuildFiles” on page 19-163
- “getSourceFiles” on page 19-165
- “getSourcePaths” on page 19-167
- “packNGo” on page 19-168
- “updateFilePathsAndExtensions” on page 19-170
- “updateFileSeparator” on page 19-171

Use these functions to access or write data to the build information object. Typically, the syntax is:

```
buildInfo.function_name(input_param1, ..., input_paramn)
```

addCompileFlags

Purpose. Add compiler options to project’s build information

Syntax.

```
addCompileFlags(buildinfo, options, groups)
```

groups is optional.

Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

options

A character array or cell array of character arrays that specifies the compiler options to be added to the build information. The function adds each option to the end of a compiler option vector. If you specify multiple options within a single character array, for example ‘-Zi -Wall’, the function adds the string to the vector as a single element. For example,

if you add `'-Zi -Wall'` and then `'-O3'`, the vector consists of two elements, as shown below.

```
'-Zi -Wall'    '-O3'
```

groups (optional)

A character array or cell array of character arrays that groups specified compiler options. You can use groups to

- Document the use of specific compiler options
- Retrieve or apply collections of compiler options

You can apply

- A single group name to one or more compiler options
- Multiple group names to collections of compiler options (available for nonmakefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to compiler options	Character array.
Apply different group names to compiler options	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>options</i> .

Description. The `addCompileFlags` function adds specified compiler options to the project's build information. MATLAB Coder stores the compiler options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *options* arguments, you can use an optional *groups* argument to group your options.

addDefines

Purpose. Add preprocessor macro definitions to project's build information

Syntax.

`addDefines(buildinfo, macrodefs, groups)`

groups is optional.

Arguments.*buildinfo*

Build information stored in `RTW.BuildInfo`.

macrodefs

A character array or cell array of character arrays that specifies the preprocessor macro definitions to be added to the object. The function adds each definition to the end of a compiler option vector. If you specify multiple definitions within a single character array, for example `'-DRT -DDEBUG'`, the function adds the string to the vector as a single element. For example, if you add `'-DPROTO -DDEBUG'` and then `'-DPRODUCTION'`, the vector consists of two elements, as shown below.

```
'-DPROTO -DDEBUG'    '-DPRODUCTION'
```

groups (optional)

A character array or cell array of character arrays that groups specified definitions. You can use groups to

- Document the use of specific macro definitions
- Retrieve or apply groups of macro definitions

You can apply

- A single group name to one or more macro definitions
- Multiple group names to collections of macro definitions (available for nonmakefile build environments only)

To...	Specify groups as a...
Apply one group name to macro definitions	Character array.
Apply different group names to macro definitions	Cell array of character arrays such that the number of group names matches the number elements you specify for <i>macrodefs</i> .

Description. The `addDefines` function adds specified preprocessor macro definitions to the project's build information. The MATLAB Coder software stores the definitions in a vector. The function adds definitions to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *macrodefs* arguments, you can use an optional *groups* argument to group your options.

addIncludeFiles

Purpose. Add include files to project's build information

object

Syntax.

`addIncludeFiles(buildinfo, filenames, paths, groups)`

paths and *groups* are optional.

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

filenames

A character array or cell array of character arrays that specifies names of include files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are `'*.*'`, `'*.h'`, and `'*.h*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate include file entries that

- You specify as input
- Already exist in the include file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

paths (optional)

A character array or cell array of character arrays that specifies paths to the include files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)

A character array or cell array of character arrays that groups specified include files. You can use groups to

- Document the use of specific include files
- Retrieve or apply groups of include files

You can apply

- A single group name to an include file
- A single group name to multiple include files
- Multiple group names to collections of multiple include files

To...	Specify groups as a...
Apply one group name to include files	Character array.
Apply different group names to include files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

Description. The `addIncludeFiles` function adds specified include files to the project's build information. The MATLAB Coder software stores the include files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the include files it adds to the build information
Cell array of character arrays	Pairs each character array with a specified include file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string (' ') for *paths*.

addIncludePaths

Purpose. Add include paths to project's build information

Syntax.

`addIncludePaths(buildinfo, paths, groups)`

groups is optional.

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

paths

A character array or cell array of character arrays that specifies include file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

The function removes duplicate include file entries that

- You specify as input
- Already exist in the include path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

groups (optional)

A character array or cell array of character arrays that groups specified include paths. You can use groups to

- Document the use of specific include paths
- Retrieve or apply groups of include paths

You can apply

- A single group name to an include path
- A single group name to multiple include paths
- Multiple group names to collections of multiple include paths

To...	Specify <i>groups</i> as a...
Apply one group name to include paths	Character array.
Apply different group names to include paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>paths</i> .

Description. The `addIncludePaths` function adds specified include paths to the project's build information. The MATLAB Coder software stores the include paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the include paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified include path. Thus, the length of the cell array must match the length of the cell array you specify for <i>paths</i> .

addLinkFlags

Purpose. Add link options to project's build information

Syntax.

`addLinkFlags(buildinfo, options, groups)`

groups is optional.

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

options

A character array or cell array of character arrays that specifies the linker options to be added to the build information. The function adds each option to the end of a linker option vector. If you specify multiple options within a single character array, for example `'-MD -Gy'`, the function adds the string to the vector as a single element. For example, if you add `'-MD -Gy'` and then `'-T'`, the vector consists of two elements, as shown below.

```
'-MD -Gy'    '-T'
```

groups (optional)

A character array or cell array of character arrays that groups specified linker options. You can use groups to

- Document the use of specific linker options

- Retrieve or apply groups of linker options

You can apply

- A single group name to one or more linker options
- Multiple group names to collections of linker options (available for nonmakefile build environments only)

To...	Specify groups as a...
Apply one group name to linker options	Character array.
Apply different group names to linker options	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>options</i> .

Description. The `addLinkFlags` function adds specified linker options to the project's build information. The MATLAB Coder software stores the linker options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *options* arguments, you can use an optional *groups* argument to group your options.

addLinkObjects

Purpose. Add link objects to project's build information

Syntax.

```
addLinkObjects(buildinfo, linkobjs, paths, priority,  
precompiled, linkonly, groups)
```

The arguments except *buildinfo*, *linkobjs*, and *paths* are optional. If you specify an optional argument, you must specify the optional arguments preceding it.

Arguments.*buildinfo*

Build information stored in `RTW.BuildInfo`.

linkobjs

A character array or cell array of character arrays that specifies the filenames of linkable objects to be added to the build information. The function adds the filenames that you specify in the function call to a vector that stores the object filenames in priority order. If you specify multiple objects that have the same priority (see *priority* below), the function adds them to the vector based on the order in which you specify the object filenames in the cell array.

The function removes duplicate link objects that

- You specify as input
- Already exist in the linkable object filename vector
- Have a path that matches the path of a matching linkable object filename

A duplicate entry consists of an exact match of a path string and corresponding linkable object filename.

paths

A character array or cell array of character arrays that specifies paths to the linkable objects. If you specify a character array, the path string applies to all linkable objects.

priority (optional)

A numeric value or vector of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority. The default priority is 1000.

precompiled (optional)

The logical value `true` or `false` or a vector of logical values that indicates whether each specified link object is precompiled.

Specify `true` if the link object has been prebuilt for faster compiling and linking and exists in a specified location.

If `precompiled` is `false` (the default), the MATLAB Coder build process creates the link object in the build folder.

This argument is ignored if `linkonly` equals `true`.

linkonly (optional)

The logical value `true` or `false` or a vector of logical values that indicates whether each specified link object is to be used only for linking.

Specify `true` if the MATLAB Coder build process should not build, nor generate rules in the makefile for building, the specified link object, but should include it when linking the final executable. For example, you can use this to incorporate link objects for which source files are not available. If `linkonly` is `true`, the value of `precompiled` is ignored.

If `linkonly` is `false` (the default), rules for building the link objects are added to the makefile. In this case, the value of `precompiled` determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (`true`) or `START_EXPAND_LIBRARIES` (`false`).

groups (optional)

A character array or cell array of character arrays that groups specified link objects. You can use groups to

- Document the use of specific link objects
- Retrieve or apply groups of link objects

You can apply

- A single group name to a linkable object
- A single group name to multiple linkable objects
- Multiple group name to collections of multiple linkable objects

To...	Specify <i>groups</i> as a...
Apply one group name to link objects	Character array.
Apply different group names to link objects	Cell array of character arrays such that the number of group names matches the number elements you specify for <i>linkobjs</i> .

The default value of *groups* is { ' ' }.

Description. The `addLinkObjects` function adds specified link objects to the project's build information. The MATLAB Coder software stores the link objects in a vector in relative priority order. If multiple objects have the same priority or you do not specify priorities, the function adds the objects to the vector based on the order in which you specify them.

In addition to the required *buildinfo*, *linkobjs*, and *paths* arguments, you can specify the optional arguments *priority*, *precompiled*, *linkonly*, and *groups*. You can specify *paths* and *groups* as a character array or a cell array of character arrays.

If You Specify <i>paths</i> or <i>groups</i> as a...	The Function...
Character array	Applies the character array to the objects it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified object. Thus, the length of the cell array must match the length of the cell array you specify for <i>linkobjs</i> .

Similarly, you can specify *priority*, *precompiled*, and *linkonly* as a value or vector of values.

If You Specify <i>priority</i>, <i>precompiled</i>, or <i>linkonly</i> as a...	The Function...
Value	Applies the value to the objects it adds to the build information.
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for <i>linkobjs</i> .

If you choose to specify an optional argument, you must specify the optional arguments preceding it. For example, to specify that objects are precompiled using the *precompiled* argument, you must specify the *priority* argument that precedes *precompiled*. You could pass the default priority value 1000, as shown below.

```
addLinkObjects(myBuildInfo, 'test1', '/proj/lib/lib1', 1000, true);
```

addNonBuildFiles

Purpose. Add nonbuild-related files to project's build information

Syntax.

```
addNonBuildFiles(buildinfo, filenames, paths, groups)
```

paths and *groups* are optional.

Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

filenames

A character array or cell array of character arrays that specifies names of nonbuild-related files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are '*.*', '*.DLL', and '*.D*'.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate nonbuild file entries that

- Already exist in the nonbuild file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

paths (optional)

A character array or cell array of character arrays that specifies paths to the nonbuild files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)

A character array or cell array of character arrays that groups specified nonbuild files. You can use groups to

- Document the use of specific nonbuild files
- Retrieve or apply groups of nonbuild files

You can apply

- A single group name to a nonbuild file
- A single group name to multiple nonbuild files
- Multiple group names to collections of multiple nonbuild files

To...	Specify groups as a...
Apply one group name to nonbuild files	Character array.
Apply different group names to nonbuild files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

Description. The `addNonBuildFiles` function adds specified nonbuild-related files, such as DLL files required for a final executable, or a README file, to the project's build information. The MATLAB Coder software stores the nonbuild files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the nonbuild files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified nonbuild file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string ('') for *paths*.

addSourceFiles

Purpose. Add source files to project's build information

Syntax.

```
addSourceFiles(buildinfo, filenames, paths, groups)
```

paths and *groups* are optional.

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

filenames

A character array or cell array of character arrays that specifies names of the source files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are '*.*', '*.c', and '*.c*'.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

paths (optional)

A character array or cell array of character arrays that specifies paths to the source files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)

A character array or cell array of character arrays that groups specified source files. You can use groups to

- Document the use of specific source files
- Retrieve or apply groups of source files

You can apply

- A single group name to a source file
- A single group name to multiple source files
- Multiple group names to collections of multiple source files

To...	Specify <i>group</i> as a...
Apply one group name to source files	Character array.
Apply different group names to source files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

Description. The `addSourceFiles` function adds specified source files to the project's build information. The MATLAB Coder software stores the source files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the source files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string ('') for *paths*.

addSourcePaths

Purpose. Add source paths to project's build information

Syntax.

```
addSourcePaths(buildinfo, paths, groups)
```

groups is optional.

Arguments.*buildinfo*

Build information stored in `RTW.BuildInfo`.

paths

A character array or cell array of character arrays that specifies source file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

Note The MATLAB Coder software does not check whether a specified path string is valid.

groups (optional)

A character array or cell array of character arrays that groups specified source paths. You can use groups to

- Document the use of specific source paths
- Retrieve or apply groups of source paths

You can apply

- A single group name to a source path
- A single group name to multiple source paths
- Multiple group names to collections of multiple source paths

To...	Specify <i>groups</i> as a...
Apply one group name to source paths	Character array.
Apply different group names to source paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>paths</i> .

Description. The `addSourcePaths` function adds specified source paths to the project's build information. The MATLAB Coder software stores the source paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument . You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the source paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source path. Thus, the length of the character array or cell array must match the length of the cell array you specify for <i>paths</i> .

Note The MATLAB Coder software does not check whether a specified path string is valid.

addTMFTokens

Purpose. Add template makefile (TMF) tokens that provide build-time information for makefile generation

Syntax.

```
addTMFTokens(buildinfo, tokennames, tokenvalues, groups)
```

groups is optional.

Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

tokennames

A character array or cell array of character arrays that specifies names of TMF tokens (for example, '|>CUSTOM_OUTNAME<|') to be added to the build information. The function adds the token names to the end of a vector in the order that you specify them.

If you specify a token name that already exists in the vector, the first instance takes precedence and its value used for replacement.

tokenvalues

A character array or cell array of character arrays that specifies TMF token values corresponding to the previously-specified TMF token names. The function adds the token values to the end of a vector in the order that you specify them.

groups (optional)

A character array or cell array of character arrays that groups specified TMF tokens. You can use groups to

- Document the use of specific TMF tokens
- Retrieve or apply groups of TMF tokens

You can apply

- A single group name to a TMF token
- A single group name to multiple TMF tokens

- Multiple group names to collections of multiple TMF tokens

To...	Specify groups as a...
Apply one group name to TMF tokens	Character array.
Apply different group names to TMF tokens	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>tokennames</i> .

Description. Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled appropriately in the template makefile (TMF) for the target selected for your project. For example, if your post code generation command calls `addTMFTokens` to add a TMF token named `|>CUSTOM_OUTNAME<|` that specifies an output file name for the build, the TMF must act on the value of `|>CUSTOM_OUTNAME<|` to achieve the desired result.

The `addTMFTokens` function adds specified TMF token names and values to the project's build information. The MATLAB Coder software stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

In addition to the required *buildinfo*, *tokennames*, and *tokenvalues* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the TMF tokens it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified TMF token. Thus, the length of the cell array must match the length of the cell array you specify for <i>tokennames</i> .

findIncludeFiles

Purpose. Find and add include (header) files to build information object

Syntax.

```
findIncludeFiles(buildinfo, extPatterns)
```

extPatterns is optional.

Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

extPatterns (optional)

A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern

- Must start with *.
- Can include a combination of alphanumeric and underscore () characters

The default pattern is *.h.

Examples of valid patterns include

```
*.h  
*.hpp  
*.x*
```

Description. The findIncludeFiles function

- Searches for include files, based on specified file name extension patterns, in the source and include paths recorded in a project's build information object
- Adds the files found, along with their full paths, to the build information object
- Deletes duplicate entries

getCompileFlags

Purpose. Compiler options from project's build information

Syntax.

```
options = getCompileFlags(buildinfo, includeGroups,  
excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Input Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of compiler flags you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of compiler flags you do not want the function to return.

Output Arguments. Compiler options stored in the project's build information.

Description. The `getCompileFlags` function returns compiler options stored in the project's build information. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of options the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (' ') for *includeGroups*.

getDefines

Purpose. Preprocessor macro definitions from project's build information

Syntax.

```
[macrodefs, identifiers, values] = getDefines(buildinfo,  
includeGroups, excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Input Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of macro definitions you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of macro definitions you do not want the function to return.

Output Arguments. Preprocessor macro definitions stored in the project's build information. The function returns the macro definitions in three vectors.

Vector	Description
<i>macrodefs</i>	Complete macro definitions with -D prefix
<i>identifiers</i>	Names of the macros
<i>values</i>	Values assigned to the macros (anything specified to the right of the first equals sign) ; the default is an empty string (' ')

Description. The `getDefines` function returns preprocessor macro definitions stored in the project's build information. When the function returns a definition, it automatically

- Prepends a -D to the definition if the -D was not specified when the definition was added to the build information
- Changes a lowercase -d to -D

Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of definitions the function is to return.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (' ') for *includeGroups*.

getFullFileList

Purpose. All files from project's build information

Syntax.

```
[fPathNames, names] = getFullFileList(buildinfo, fcase)
```

fcase is optional.

Input Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

fcase (optional)

The string 'source', 'include', or 'nonbuild'. If the argument is omitted, the function returns all the files from the build information object.

If You Specify...	The Function...
'source'	Returns source files from the build information object.
'include'	Returns include files from the build information object.
'nonbuild'	Returns nonbuild files from the build information object.

Output Arguments. Fully-qualified file paths and file names for files stored in the project's build information.

Note Usually it is unnecessary to resolve the path of every file in the project build information, because the makefile for the project build will resolve file locations based on source paths and rules. Therefore, `getFullFileList` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

Description. The `getFullFileList` function returns the fully-qualified paths and names of all files, or files of a selected type (source, include, or nonbuild), stored in the project's build information.

getIncludeFiles

Purpose. Get include files from project's build information

Syntax.

```
files = getIncludeFiles(buildinfo, concatenatePaths,  
replaceMatlabroot, includeGroups, excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

concatenatePaths

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Concatenates and returns each filename with its corresponding path.
<code>false</code>	Returns only filenames.

replaceMatlabroot

The logical value `true` or `false`.

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of include files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of include files you do not want the function to return.

Returns. Names of include files stored in the project's build information.

Description. The `getIncludeFiles` function returns the names of include files stored in the project's build information. Use the `concatenatePaths` and `replaceMatlabroot` arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of include files the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string (' ') for `includeGroups`.

getIncludePaths

Purpose. Get include paths from project's build information

Syntax.

```
files=getIncludePaths(buildinfo, replaceMatlabroot,
includeGroups, excludeGroups)
```

`includeGroups` and `excludeGroups` are optional.

Input Arguments.*buildinfo*

Build information stored in RTW.BuildInfo.

replaceMatlabroot

The logical value true or false.

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of include paths you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of include paths you do not want the function to return.

Output Arguments. Paths of include files stored in the build information object.**Description.** The `getIncludePaths` function returns the names of include file paths stored in the project's build information. Use the *replaceMatlabroot* argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of include file paths the function returns.If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (' ') for *includeGroups*.**getLinkFlags****Purpose.** Link options from project's build information

Syntax.

```
options=getLinkFlags(buildinfo, includeGroups, excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Input Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

includeGroups (optional)

A character array or cell array that specifies groups of linker flags you want the function to return.

excludeGroups (optional)

A character array or cell array that specifies groups of linker flags you do not want the function to return. To exclude groups and not include specific groups, specify an empty cell array (' ') for *includeGroups*.

Output Arguments. Linker options stored in the project's build information.

Description. The `getLinkFlags` function returns linker options stored in the project's build information. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of options the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (' ') for *includeGroups*.

getNonBuildFiles

Purpose. Nonbuild-related files from project's build information

Syntax.

```
files=getNonBuildFiles(buildinfo, concatenatePaths,  
replaceMatlabroot, includeGroups, excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Input Arguments.*buildinfo*Build information stored in `RTW.BuildInfo`.*concatenatePaths*The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Concatenates and returns each filename with its corresponding path.
<code>false</code>	Returns only filenames.

*replaceMatlabroot*The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
<code>false</code>	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you do not want the function to return.

Output Arguments. Names of nonbuild files stored in the project's build information.

Description. The `getNonBuildFiles` function returns the names of nonbuild-related files, such as DLL files required for a final executable, or a README file, stored in the project's build information. Use the `concatenatePaths` and `replaceMatlabroot` arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of nonbuild files the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string (' ') for `includeGroups`.

getSourceFiles

Purpose. Source files from project's build information

Syntax.

```
srcfiles=getSourceFiles(buildinfo, concatenatePaths,
replaceMatlabroot, includeGroups, excludeGroups)
```

`includeGroups` and `excludeGroups` are optional.

Input Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

concatenatePaths

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Concatenates and returns each filename with its corresponding path.
<code>false</code>	Returns only filenames.

Note Usually it is unnecessary to resolve the path of every file in the project build information, because the makefile for the project build will resolve file locations based on source paths and rules. Therefore, specifying `true` for `concatenatePaths` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

replaceMatlabroot

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
<code>false</code>	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of source files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of source files you do not want the function to return.

Output Arguments. Names of source files stored in the project's build information.

Description. The `getSourceFiles` function returns the names of source files stored in the project's build information. Use the `concatenatePaths` and `replaceMatlabroot` arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of source files the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string (`' '`) for `includeGroups`.

getSourcePaths

Purpose. Source paths from project's build information

Syntax.

```
files=getSourcePaths(buildinfo, replaceMatlabroot,
includeGroups, excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Input Arguments.

buildinfo

Build information stored in RTW.BuildInfo.

replaceMatlabroot

The logical value true or false.

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of source paths you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of source paths you do not want the function to return.

Output Arguments. Paths of source files stored in the project's build information.

Description. The `getSourcePaths` function returns the names of source file paths stored in the project build information. Use the `replaceMatlabroot` argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of source file paths that the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string (' ') for `includeGroups`.

packNGo

Purpose. Package generated code in zip file for relocation

Syntax.

`packNGo(buildinfo, propVals...)`

`propVals` is optional.

Input Arguments. Arguments

`buildinfo`

Build information loaded from the build folder.

`propVals` (optional)

A cell array of property-value pairs that specify packaging details.

To...	Specify Property...	With Value...
Package all generated code files in a zip file as a single, flat folder	'packType'	'flat' (default)
Package generated code files hierarchically in a primary zip file that contains three secondary zip files: <ul style="list-style-type: none"> • <code>mlrFiles.zip</code> — files in your <code>matlabroot</code> folder tree • <code>sDirFiles.zip</code> — files in and under your build folder 	'packType'	'hierarchical' Paths for files in the secondary zip files are relative to the root folder of the primary zip file.

To...	Specify Property...	With Value...
<ul style="list-style-type: none"> • <code>otherFiles.zip</code> — required files not in the <code>matlabroot</code> or <code>start</code> folder trees 		
Specify a file name for the primary zip file	'fileName'	'string' Default: 'untitled.zip' If you omit the .zip file extension, the function adds it.
Include only the minimal header files required to build the code in the zip file	'minimalHeaders'	true (default)
Include all header files found on the include path in the zip file	'minimalHeaders'	false

Description. The `packNGo` function packages the following code files in a compressed zip file so you can relocate, unpack, and rebuild them in another development environment.

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- Nonbuild-related files (for example, `.dll` files required for a final executable file and `.txt` informational files)
- MAT-file that contains the build information object (`.mat` file)

Use this function to relocate files so that they can be recompiled for a specific target environment, or rebuilt in a development environment in which MATLAB is not installed.

By default, the `packNGo` function creates a zip file, `foo.zip`, in the current working folder.

By default, the function packages the files as a flat folder structure in a zip file. You can customize the output by specifying property name and value pairs as previously described.

After relocating the zip file, use a standard zip utility to unpack the compressed file.

packNGo Function Limitations. The following limitations apply to use of the packNGo function:

- The function operates on source files only, such as *.c, *.cpp, and *.h files. The function does not support compile flags, defines, or makefiles.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if they are not used.
- packNGo does not package the code generated for MEX targets.

See Also.

- “Package Generated Code at the Command Line” on page 19-198
- “Package Code For Use in Another Development Environment” on page 19-196

updateFilePathsAndExtensions

Purpose. Update files in project build information with missing paths and file extensions

Syntax.

`updateFilePathsAndExtensions(buildinfo, extensions)`

extensions is optional.

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

extensions (optional)

A cell array of character arrays that specifies the extensions (file types) of files for which to search and include in the update processing. By default, the function searches for files with a .c extension. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For example,

if you specify `{'.c' '.cpp'}`, and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.

Description. Using paths that already exist in a project's build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information need to be updated with a path or file extension. This function can be particularly useful for

- Maintaining build information for a toolchain that requires the use of file extensions
- Updating multiple customized instances of build information for a given project

updateFileSeparator

Purpose. Change file separator used in project's build information

Syntax.

`updateFileSeparator(buildinfo, separator)`

Arguments.

buildinfo

Build information stored in `RTW.BuildInfo`.

separator

A character array that specifies the file separator `\` (Windows) or `/` (UNIX) to be applied to file path specifications.

Description. The `updateFileSeparator` function changes instances of the current file separator (`/` or `\`) in a project's build information to the specified file separator.

The default value for the file separator matches the value returned by the MATLAB command `filesep`. For makefile based builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile. If the `GenerateMakefile` parameter is set, the MATLAB Coder software overrides the default separator and updates the

build information after evaluating the `PostCodeGenCommand` configuration parameter.

Programming a Post-Code-Generation Command

A post-code-generation command is a MATLAB file that typically calls functions that get data from or add data to the build information object. For example, you can access the project name in the variable `projectName` and the `RTW.BuildInfo` object in the variable `buildInfo`. You can program the command as a script or a function.

If You Program the Command as a...	Then the...
Script	Script can gain access to the project (top-level function) name and the build information directly.
Function	Function can pass the project name and the build information as arguments.

If your post-code-generation command calls user-defined functions, make sure that the functions are on the MATLAB path. If the build process cannot find a function that you use in your command, the process fails.

You can call combinations of build information functions to customize the post-code-generation build. See “Programming and Using a Post-Code-Generation Command at the Command Line” on page 19-174

Using a Post-Code-Generation Command in Your Build

After you program a post-code-generation command, you must include this command in the build processing. You can include the command from the project settings dialog box or the command line.

Including a Post-Code-Generation Command in the Project Settings Dialog Box.

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On this tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the Project Settings dialog box, click the **Custom Code** tab.
- 4 On this tab, set the **Post-code-generation command** parameter. Close the dialog box.

How you use the `PostCodeGenCommand` option depends on whether you program the command as a script or a function. See “Including a Post-Code-Generation Command at the Command Line” on page 19-173 and “Including a Post-Code-Generation Command in the Project Settings Dialog Box.” on page 19-173.

Including a Post-Code-Generation Command at the Command Line

Set the `PostCodeGenCommand` option for the code generation configuration object (`coder.MexCodeConfig`, `coder.CodeConfig` or `coder.EmbeddedCodeConfig`).

How you use the `PostCodeGenCommand` option depends on whether you program the command as a script or a function. See “Including a Post-Code-Generation Command at the Command Line” on page 19-173 and “Including a Post-Code-Generation Command in the Project Settings Dialog Box.” on page 19-173.

Programming the Post-Code-Generation Command as a Script

Set `PostCodeGenCommand` to the script name.

At the command line, enter:

```
cfg = coder.config('lib');  
cfg.PostCodeGenCommand = 'ScriptName';
```

Programming the Post-Code-Generation Command as a Function

Set `PostCodeGenCommand` to the function signature. When you define the command as a function, you can specify an arbitrary number of input arguments. If you want to access the project name, include `projectName` as an argument. If you want to modify or access build information, add `buildInfo` as an argument.

At the command line, enter:

```
cfg = coder.config('lib');
cfg.PostCodeGenCommand = 'FunctionName(projectName, buildInfo)';
```

Programming and Using a Post-Code-Generation Command at the Command Line

The following example shows how to program and use a post-code-generation command as a function. The `setbuildargs` function takes the build information object as a parameter, sets up link options, and adds them to the build information object.

- 1 Create a post-code-generation command as a function, `setbuildargs`, which takes the `buildInfo` object as a parameter:

```
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);
```

- 2 Create a code generation configuration object. Set the `PostCodeGenCommand` option to `'setbuildargs(buildInfo)'` so that this command is included in the build processing:

```
cfg = coder.config('mex');
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';
```

- 3 Using the `-config` option, generate a MEX function passing the configuration object to `codegen`. For example, for the function `foo` that has no input parameters:


```
codegen -config cfg foo
```

Code Generation Reports

In this section...
“About Code Generation Reports” on page 19-176
“Enable Code Generation Reports” on page 19-179
“View Your MATLAB Code in a Report” on page 19-180
“Viewing Call Stack Information” on page 19-181
“View Generated C/C++ Code in a Report” on page 19-184
“Viewing the Build Summary Information” on page 19-184
“View Error and Warning Messages in a Report” on page 19-185
“Viewing Variables in Your MATLAB Code” on page 19-186
“Viewing Target Build Information” on page 19-192
“Keyboard Shortcuts for the Code Generation Report” on page 19-193
“Report Limitations” on page 19-193

About Code Generation Reports

At code-generation time, MATLAB Coder produces reports to help you debug your MATLAB code and to verify that your MATLAB code is suitable for code generation.

Report Generation

If MATLAB Coder detects errors or warnings, the software automatically produces a code generation report. You can also use an option to request reports even if MATLAB Coder does not detect errors or warnings.

The report provides links to your MATLAB code and C/C++ code files. It also provides compile-time type information for the variables and expressions in your MATLAB code. This information simplifies finding sources of error messages and aids understanding of type propagation rules.

Names and Locations of Code Generation Reports

MATLAB Coder produces code generation reports in the following locations. The top-level html file at each location is `index.html`.

- For MEX functions:

```
output_folder  
/mex/primary_function_name/html
```

- For C/C++ executables:

```
output_folder/exe/primary_function_name/html
```

- For C/C++ libraries:

```
output_folder/lib/primary_function_name/html
```

Note The default output folder is `codegen`, but you can specify a different folder. For more information, see “Specify Output File Locations” on page 16-41.

Opening Code Generation Reports

Opening Code Generation Reports in the Project Interface. On the project **Build** tab, the **Build Results** pane provides information about the most recent build. If the code generation report is enabled or build errors occur, MATLAB Coder generates a report that provides detailed information about the most recent build and provides a link to the report.

To view the report, click the **View report** link. After a build completes, this report provides links to your MATLAB code and generated C/C++ files as well as compile-time type information for the variables in your MATLAB code. If build errors occur, it lists errors and warnings.

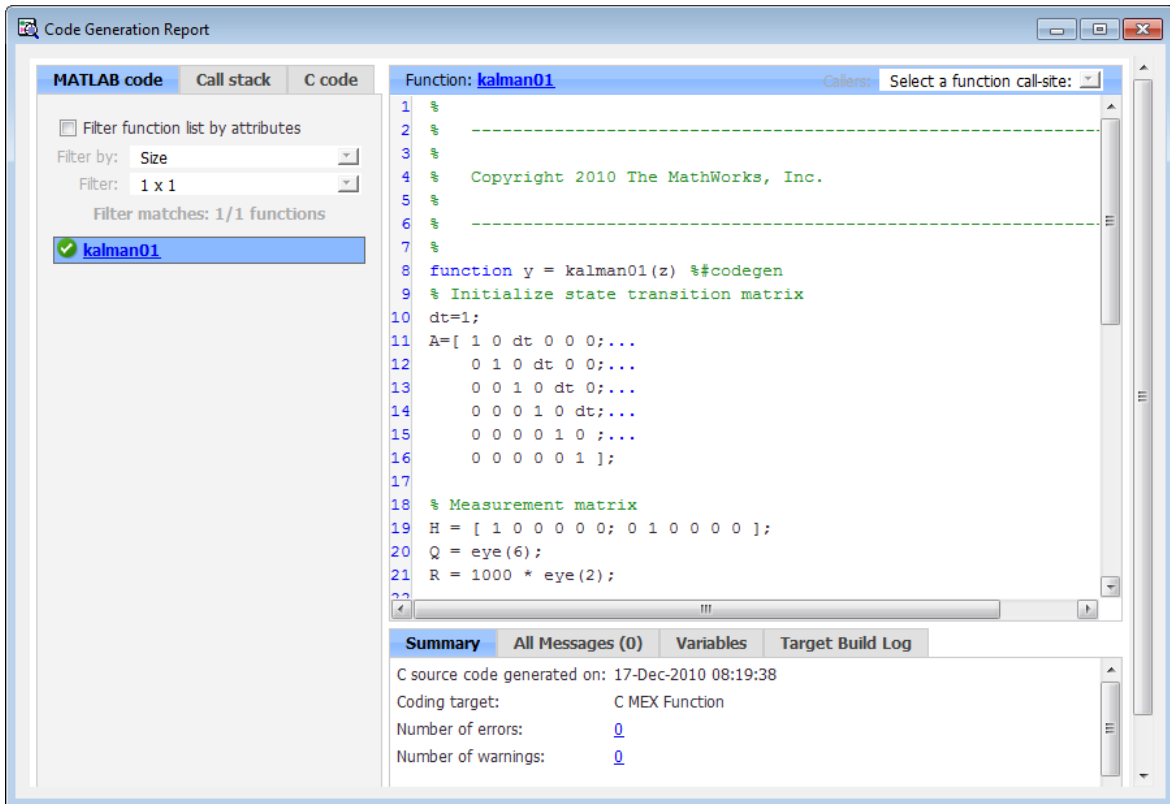
Opening Code Generation Reports at the Command Line. If you specify the `-launchreport` option, the code generation report opens automatically.

If MATLAB Coder did not detect build errors, to open the code generation report, in the MATLAB Command Window, click the **View report** link.

If MATLAB Coder detected build errors, to open the error report, in the MATLAB Command Window, click the **Open error report** link.

Description of Code Generation Reports

When you generate code for MATLAB files from a MATLAB Coder project, or from the command line using the `codegen -report` option, MATLAB Coder generates a report. The following example shows a report for a completed build.



The report provides the following information, as applicable:

- MATLAB code information, including a list of functions and classes and their build status
- Call stack information, providing information on the nesting of function calls
- Links to generated C/C++ code files
- Summary of build results, including type of target and number of warnings or errors
- List of error and warning messages
- List of variables in your MATLAB code
- Target build log that records compilation and linking activities

Enable Code Generation Reports

How to Enable Code Generation Reports in the Project Settings Dialog Box

- 1 On the project **Build** tab, click the **More settings** link.
- 2 In the **Project Settings** dialog box, click the **Debugging** tab.
- 3 On the **Debugging** tab, check **Always create a code generation report**.

If you want the code generation or error report to open automatically when MATLAB Coder finishes building a project, check **Automatically launch a report if one is generated**.

How to Enable Code Generation Reports at the Command Line

Use the `codegen` function `-report` option. To generate a standalone C/C++ static library and code generation report for a function `foo` that has no input parameters, at the MATLAB command line, enter:




```
codegen -config:lib -report foo
```

If you want the code generation or error report to open automatically, use the `-launchreport` option instead of the `-report` option.

View Your MATLAB Code in a Report

To view your MATLAB code, click the **MATLAB code** tab. The code generation report displays the code for the function or class highlighted in the list on this tab.

The **MATLAB code** tab provides:

- A list of the MATLAB functions and classes that have been built. Depending on the build results, the report displays icons next to each function or class name:
 -  Errors in function or class.
 -  Warnings in function or class.
 -  Completed build, no errors or warnings.
- A filter control. You can use **Filter functions and methods** to sort your functions and methods by:
 - Size
 - Complexity
 - Class

Viewing Local Functions

The code generation report annotates the local function with the name of the parent function in the list of functions on the **MATLAB code** tab.

For example, if the MATLAB function `fcn1` contains the local function `local_fcn1`, and `fcn2` contains the local function `local_fcn2`, the report displays:

```
fcn1 > local_fcn1  
fcn2 > local_fcn2
```

Viewing Specializations

If your MATLAB function calls the same function with different types of inputs, the code generation report numbers each of these **specializations** in the list of functions on the **MATLAB code** tab.

For example, if the function `fcn` calls the function `subfcn` with different types of inputs:

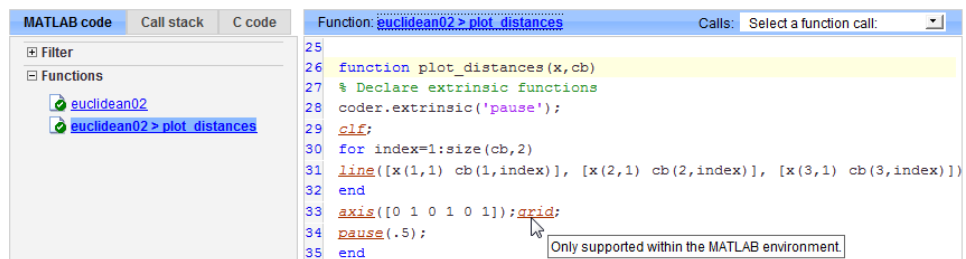
```
function y = fcn(u) %#codegen
% Specializations
y = y + subfcn(single(u));
y = y + subfcn(double(u));
```

The code generation report numbers the specializations in the list of functions:

```
fcn > subfcn > 1
fcn > subfcn > 2
```

Viewing Extrinsic Functions

The report highlights the extrinsic functions that are supported only within the MATLAB environment.



The screenshot shows the MATLAB code generation report interface. On the left, the 'MATLAB code' tab is active, displaying a list of functions under 'Functions'. The function 'euclidean02 > plot_distances' is selected. On the right, the code for this function is displayed, with line numbers 25 to 35. The code includes a function definition, a comment '% Declare extrinsic functions', a call to 'coder.extrinsic('pause')', and a loop that calls 'line' and 'axis'. The 'axis' call is highlighted with a tooltip that says 'Only supported within the MATLAB environment'.

```
25
26 function plot_distances(x,cb)
27 % Declare extrinsic functions
28 coder.extrinsic('pause');
29 clf;
30 for index=1:size(cb,2)
31 line([x(1,1) cb(1,index)], [x(2,1) cb(2,index)], [x(3,1) cb(3,index)])
32 end
33 axis([0 1 0 1 0 1]);axis;
34 pause(.5);
35 end
```

Viewing Call Stack Information

The code generation report provides call stack information:

- On the **Call stack** tab.
- In the list of **Calls** at the top right of the report.

This list shows the calls from and to the function or method. If a function is called from more than one function, this list provides details of each call-site. Otherwise, the list is disabled.

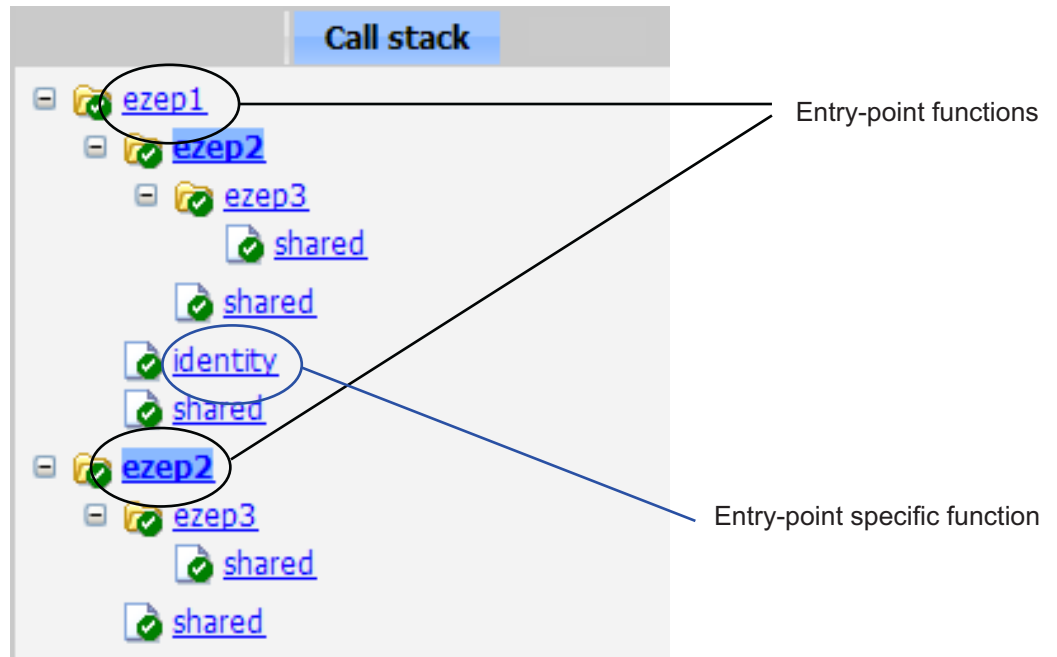
Viewing Call Stack Information on the Call stack Tab

To view call stack information, click the **Call stack** tab.

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

For more than one entry-point function, the call stack displays a separate tree for each entry point. You can easily distinguish between shared and entry-point specific functions. If you click a shared function, the report highlights instances of this function. If you click an entry-point specific function, the report highlights only that instance.

For example, in the following call stack, `ezep1` and `ezep2` are entry-point functions. `identity` is an entry-point specific function, called only by `ezep1`. Functions `ezep3` and `shared` are shared functions.



Viewing Call Sites in the Callers List

If a function or method is called from more than one function or method, or if the function or method calls other functions or methods, the **Calls** list provides details of each call site. To navigate between call sites, select a call site from the **Calls** list. If the function is not called more than once, this list is disabled.

If you select the entry-point function `ezep2` in the call stack, the **Calls** list displays the other call-site in `ezep1`.

Function: <code>ezep2 > 1</code>	Callers: <input type="text" value="Select a function call-site:"/>
<pre> 1 function y = ezep2(u) %#codegen 2 y = shared(ezep3(u)); </pre>	<input type="text" value="Select a function call-site:"/> <input type="text" value="ezep1 at 2"/>

View Generated C/C++ Code in a Report

To view a list of the generated C/C++ files, click the **C-code** tab. The code generation report displays a list of the generated files. Click a file in the list to view the code in the code pane.

Tracing Generated Code Back to MATLAB Source Code

You can configure `codegen` to generate C code that includes the MATLAB source code as comments. In these auto-generated comments, `codegen` precedes each line of source code with a traceability tag that provides details about the location of the source code. For more information, see “Generation of Traceable Code” on page 19-88.

For code generated with an Embedded Coder license, these traceability tags are hyperlinks. Click a tag to go the relevant line in the source code in the MATLAB editor.

Navigating to C/C++ Code Source Files

When viewing C/C++ code in the code pane, click the blue link to the source file at the top of the pane to open the associated source code file in the MATLAB editor.

Viewing Type Definitions

The code generation report provides links to the definitions of data types. When viewing C/C++ code in the code pane, click the blue link for a data type to see its definition.

Viewing Custom Code

The report displays custom code with color syntax highlighting. To learn what these colors mean and how to customize color settings, see “Colors in the MATLAB Editor”.

Viewing the Build Summary Information

To view a summary of the build results, including type of target and number of errors or warnings, click the **Summary** tab.

View Error and Warning Messages in a Report

MATLAB Coder automatically reports errors and warnings. If errors occur during the build, MATLAB Coder does not generate code. The report lists the messages in the order that MATLAB Coder detects them. It is a best practice to address the first message in the list, because often subsequent errors and warnings are related to the first message. If the build produces warnings, but no errors, MATLAB Coder does generate code.

The code generation report provides information about errors and warnings by:

- Listing errors and warnings on the **All Messages** tab. The report lists these messages in chronological order.
- Highlighting errors and warnings on the **MATLAB code** pane.
- If applicable, recording compilation and linking issues on the **Target Build Log** tab. If compilation or linking errors occur, the code generation report opens with the **Target Build Log** tab selected so that you can view the build log.

Viewing Errors and Warnings in the All Messages Tab

If errors or warnings occur during the build, click the **All Messages** tab to view a complete list of these messages. The code generation report marks messages:



Error



Warning

To locate the incorrect line of code for an error or warning in the list, click the message in the list. The code generation report highlights errors in the list and MATLAB code in red and highlights warnings in orange. Click the blue line number next to the incorrect line of code in the MATLAB code pane to go to the error in the source file.

Note You can fix errors only in the source file.

Viewing Error and Warning Information in Your MATLAB Code

If errors or warnings occur during the build, the code generation report underlines them in your MATLAB code. The report underlines errors in red and underlines warnings in orange. To learn more about a particular error or warning, place your pointer over the underlined text.

Viewing Compilation and Linking Errors and Warnings

If compilation or linking errors occur, the code generation report opens with the **Target Build Log** tab selected so that you can view the build log.

Viewing Variables in Your MATLAB Code

The report provides compile-time type information for the variables and expressions in your MATLAB code, including name, type, size, complexity, and class. It also provides type information for fixed-point data types, including word length and fraction length. You can use this type information to find sources of error messages and to understand type propagation rules.

You can view information about the variables in your MATLAB code:

- On the **Variables** tab, view the list
- In your MATLAB code, place your pointer over the variable name

Viewing Variables in the Variables Tab

To view a list of the variables in your MATLAB function, click the **Variables** tab. The report displays a complete list of variables in the order that they appear in the function selected on the **MATLAB code** tab. Clicking a variable in the list highlights instances of that variable, and scrolls the MATLAB code pane so that you can view the first instance.

The report provides the following information about each variable, as applicable.

- Order
- Name
- Type

- Size
- Complexity
- Class
- DataTypeMode (DT mode) — for fixed-point data types only. For more information, see “DataTypeMode”.
- Signed — sign information for built-in data types, signedness information for fixed-point data types
- Word length (WL) — for fixed-point data types only
- Fraction length (FL) — for fixed-point data types only

Note For more information on viewing fixed-point data types, see “Create and Use Fixed-Point Code Generation Reports”.

It only displays a column if at least one variable in the code has information in that column. For example, if the code does not contain fixed-point data types, the report does not display the DT mode, WL or FL columns.

Sorting Variables in the Variables Tab. By default, the report lists the variables in the order that they appear in the selected function.

You can sort the variables by clicking the column headings on the **Variables** tab. To sort the variables by multiple columns, hold down the **Shift** key when clicking the column headings.

To restore the list to the original order, click the **Order** column heading.

Viewing Structures on the Variables Tab. You can expand structures listed on the **Variables** tab to display the field properties.

Summary		All Messages (0)	Variables			
Order	Variable	Type	Size	Complex	Class	
☐ 1	s	Output	1 x 1	-	struct	
1.1	s.a	Field	1 x 1	No	double	
1.2	s.b	Field	1 x 1	No	double	
2	a	Input	1 x 1	No	double	
3	b	Input	1 x 1	No	double	

If you sort the variables by type, size, complexity or class, a structure and its fields might not appear sequentially in the list. To restore the list to the original order, click the **Order** column heading.


Viewing Information About Variable-Size Arrays in the Variables Tab.

For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon **:**.

In the following report, variable *A* is variable-size. Its maximum computed size is 1×100.

Summary		All Messages (0)	Variables			
Order	Variable	Type	Size	Complex	Class	
1	B	Output	1 x :100	No	double	
2	A	Input	1 x :100	No	double	
3	tol	Input	1 x 1	No	double	
4	k	Local	1 x 1	No	double	
5	i	Local	1 x 1	No	double	

If the code generation software cannot compute the maximum size of a variable-size array, the report displays the size as **:?**.

Summary		All Messages (1)	Variables	
Order	Type	Function	Line	Description
1		emldemo_uniquetol	10	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [1 x :?]. This error may be reported due to a limitation of the underlying analysis.

If you declare a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends * to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions do not change during execution.

Summary		All Messages (0)	Variables	Target Build Log	
Order	Variable	Type	Size	Complex	Class
1	y	Output	1 x 10 *	No	double

For more information on how to use the size information for variable-sized arrays, see “Variable-Size Data Definition for Code Generation” on page 7-3.

Viewing Renamed Variables in the Variables Tab. If your MATLAB function reuses a variable with different size, type, or complexity, the code generation software attempts to create separate, uniquely named variables in the generated code. For more information, see “Reuse the Same Variable with Different Properties” on page 5-11. The report numbers the renamed variables in the list on the **Variables** tab. When you place your pointer over a renamed variable, the report highlights only the instances of this variable that share the same data type, size, and complexity.

For example, suppose your code uses the variable `t` in a for-loop to hold a scalar double, and reuses it outside the for-loop to hold a 5x5 matrix. The report displays two variables, `t>1` and `t>2` in the list on the **Variables** tab.

```

6  if all(all(u))
7      % First time t is used to hold a scalar double value
8      t = mean(mean(u)) / numel(u);
9      u = u - t;
10 end

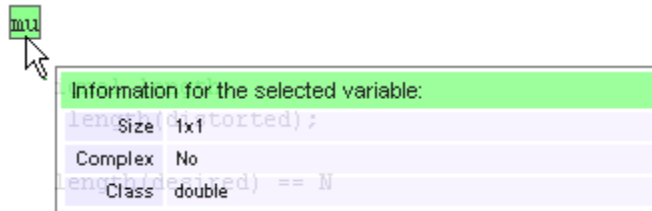
```

Summary	All Messages (0)	Variables				
Order	Variable	Type	Size	Complex	Class	
1	u	Input	5 x 5	No	double	
2	t > 1	Local	5 x 5	No	double	
3	t > 2	Local	1 x 1	No	double	

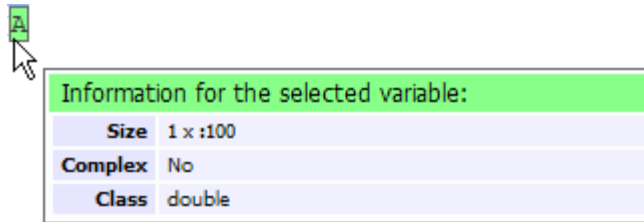
Viewing Information About Variables and Expressions in Your MATLAB Function Code

To view information about a particular variable or expression in your MATLAB function code, on the MATLAB code pane, place your pointer over the variable name or expression. The report highlights variables and expressions in different colors:

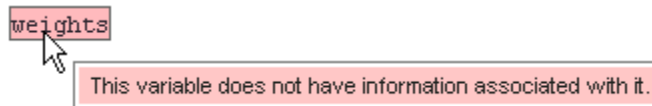
Green, when the variable has data type information at this location in the code.



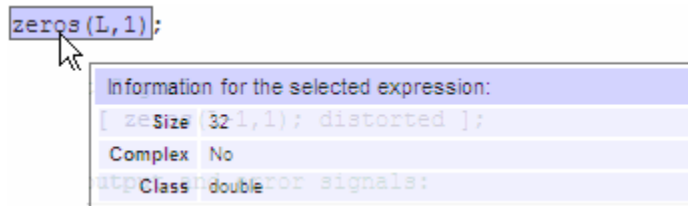
For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon `:`. Here the array `A` is variable-sized with a maximum computed size of `1 x 100`.



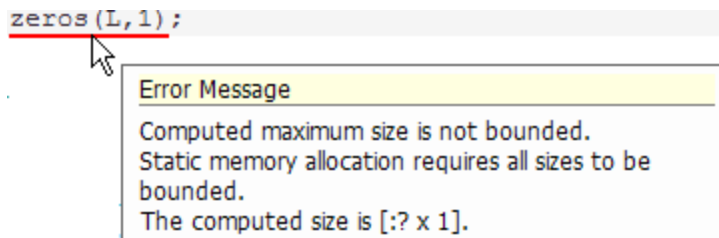
Pink, when the variable has no data type information.



Purple, information about expressions. You can also view information about expressions in your MATLAB code. On the MATLAB code pane, place your pointer over an expression. The report highlights expressions in purple and provides more detailed information.



Red, when there is error information for an expression. If the code generation software cannot compute the maximum size of a variable-size array, the report underlines the variable name and provides error information.



Viewing Target Build Information

If the build completes, MATLAB Coder provides target build information on the **Target Build Log** tab, including:

- Build folder

Clicking this link changes the MATLAB current folder to the build folder.

- Make wrapper

The batch file name that MATLAB Coder used for this build.

- Build log

If compilation or linking errors occur, the code generation report opens with the **Target Build Log** tab selected so that you can view the build log.

Summary	All Messages (12)	Variables	Target Build Log
Build Parameters			
Build directory	C:\Work\emcpri\mexfcn\warn		
Make wrapper	warn_mex.bat		
Build Log			
<pre> 1 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 2 warn_data.c 3 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 4 warn.c 5 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 6 warn_initialize.c 7 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 8 warn_terminate.c 9 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 10 morphfcn.c 11 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 12 warn_api.c 13 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 14 warn_mex.c 15 link /dll /export:mexFunction /LIBPATH:"\\.\mathworks\devel\JOBARC-1\Aslrtw\~SNAPSH\2009_0-1\current\ma 16 Creating library warn.x and object warn.exp 17 mt -outputresource:"warn_mexw32;2" -manifest "warn_mexw32.manifest" 18 Microsoft (R) Manifest Tool version 5.2.3790.2014 19 Copyright (c) Microsoft Corporation 2005. 20 All rights reserved.</pre>			

Keyboard Shortcuts for the Code Generation Report

You can use the following keyboard shortcuts to navigate between the different panes in the code generation report. Once you have selected a pane, use the **Tab** key to advance through data in that pane.

To select ...	Use...
MATLAB code Tab	Ctrl+m
Call stack Tab	Ctrl+k
C code Tab	Ctrl+c
Code Pane	Ctrl+w
Summary Tab	Ctrl+s
All Messages Tab	Ctrl+a
Variables Tab	Ctrl+v
Target Build Log Tab	Ctrl+t

Report Limitations

The report displays information about the variables and expressions in your MATLAB code with the following limitations:

varargin and varargout

The report does not support varargin and varargout arrays.

Loop Unrolling

The report does not display full information for unrolled loops. It displays data types of one arbitrary iteration.

Dead Code

The report does not display information about dead code.

Structures

The report does not provide complete information about structures.

- On the **MATLAB code** pane, the report does not provide information about all structure fields in the `struct()` constructor.
- On the **MATLAB code** pane, if a structure has a nonscalar field, and an expression accesses an element of this field, the report does not provide information for the field.

Column Headings on Variables Tab

If you scroll through the list of variables, the report does not display the column headings on the **Variables** tab.

Multiline Matrices

On the **MATLAB code** pane, the report does not support selection of multiline matrices. It supports only selection of individual lines at a time. For example, if you place your pointer over the following matrix, you cannot select the entire matrix.

```
out1 = [1 2 3;  
        4 5 6];
```

The report does support selection of single line matrices.

```
out1 = [1 2 3; 4 5 6];
```

Troubleshooting

Run-time Stack Overflow

If your C compiler reports a run-time stack overflow, set the value of the maximum stack usage parameter to be less than the available stack size. In a project, on the **Project Settings** dialog box **Memory** tab, set the **Stack usage max** parameter. For command-line configuration objects (`coder.MexCodeConfig`, `coder.CodeConfig`, `coder.EmbeddedCodeConfig`), set the `StackUsageMax` parameter.

Package Code For Use in Another Development Environment

In this section...

“When to Package Code” on page 19-196

“Package Generated Code in a Project” on page 19-196

“Package Generated Code at the Command Line” on page 19-198

When to Package Code

If you need to relocate the generated code files to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB, use either the `packNGo` function at the command line or the `package` option in a project. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

See “Package Generated Code at the Command Line” on page 19-198 and “Package Generated Code in a Project” on page 19-196.

Package Generated Code in a Project

This example shows how to package generated code into a zip file for relocation using the `Package` option in a MATLAB Coder project. By default, the zip file is created in the project folder.

- 1 In a local writable folder, for example `c:\work`, write a function `foo` that takes two double inputs.

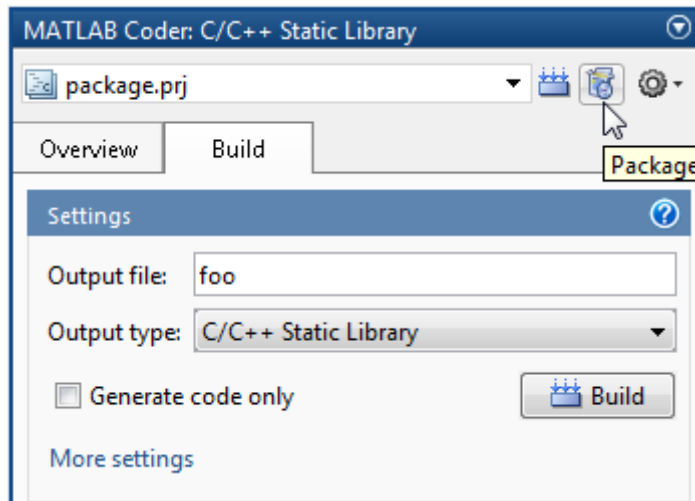
```
function y = foo(A,B)
    y = A + B;
end
```

- 2 In the same folder, create a new project.

```
coder -new package.prj
```

- 3 Add the file `foo` as an entry-point to the project.

- 4 Specify that inputs A and B are scalar doubles.
- 5 On the project **Build** tab, set **Output type** to build a static or dynamic library or executable. You cannot package the code generated for MEX targets.
- 6 At the top of the project, click **Package**.



Because you have not already built the project, MATLAB Coder builds the project.

- 7 When prompted, save the package file using the default path and file name. By default, MATLAB Coder derives the name of the package file from the project name and saves it in the current working folder. This zip file contains the C code and header files required for relocation. It does not contain compile flags, defines, or makefiles.
- 8 Inspect the contents of `package_pkg.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool you use you might be able to open and inspect the file without unpacking it.

You can now relocate the resulting zip file to the destination development environment and unpack the file.

Package Generated Code at the Command Line

This example shows how to package generated code into a zip file for relocation using the `packNGo` function at the command line.

- 1 In a local writable folder, for example `c:\work`, write a function `foo` that takes two double inputs.

```
function y = foo(A,B)
    y = A + B;
end
```

- 2 Generate a static library for function `foo`. (`packNGo` does not package MEX function code.)

```
codegen -report -config:lib foo -args {0,0}
```

`codegen` generates code in the `c:\work\codegen\lib\foo` folder.

- 3 Load the `buildInfo` object.

```
load('c:\work\codegen\lib\foo\buildInfo.mat')
```

- 4 Create the zip file.

```
packNGo(buildInfo, 'fileName', 'foo.zip');
```

Alternatively, use the notation:

```
buildInfo.packNGo('fileName', 'foo.zip');
```

The `packNGo` function creates a zip file, `foo.zip`, in the current working folder. This zip file contains the C code and header files required for relocation. It does not contain compile flags, defines, or makefiles.

In this example, you specify only the file name. Optionally, you can specify additional packaging options.

To...	Specify...
Change the structure of the file packaging to hierarchical	<code>packNGo(buildInfo, {'packType' 'hierarchical'});</code>
Change the structure of the file packaging to hierarchical and rename the primary zip file	<code>packNGo(buildInfo, {'packType' 'hierarchical'... 'fileName' 'zippedsrcs'});</code>
Include all header files found on the include path (rather than the minimal header files required to build the code) in the zip file	<code>packNGo(buildInfo, {'minimalHeaders' false});</code>

For more information, see “packNGo” on page 19-168 and “Choose a Structure for the Zip File” on page 19-199.

- 5 Inspect the contents of `foo.zip` to verify that it is ready for relocation to the destination system. Depending on the zip tool you use you might be able to open and inspect the file without unpacking it. If you need to unpack the file and you packaged the generated code files as a hierarchical structure, you will need to unpack the primary and secondary zip files. When you unpack the secondary zip files, relative paths of the files are preserved.

You can now relocate the resulting zip file to the destination development environment and unpack the file.

Choose a Structure for the Zip File

Before you generate and package the files, decide whether you want the files to be packaged in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure. This approach is the simplest and might be the optimal choice.

If...	Then Use a...
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	Single, flat folder structure
The target development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code is dependent on the relative location of files	Hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of zip files. There is a primary zip file, which in turn contains the following secondary zip files:

- `m1rFiles.zip` — files in your *matlabroot* folder tree
- `sDirFiles.zip` — files in and under your build folder where you initiated code generation
- `otherFiles.zip` — required files not in the *matlabroot* or start folder trees

Paths for the secondary zip files are relative to the root folder of the primary zip file, maintaining the source development folder structure.

Custom Toolchain Registration

- “Adding a Custom Toolchain” on page 20-2
- “Register a Toolchain” on page 20-9
- “Determine if a Toolchain is Registered” on page 20-11
- “Using a Custom Toolchain — Command Line Approach” on page 20-12
- “Using a Custom Toolchain — UI Approach” on page 20-13
- “Removing a Toolchain” on page 20-14
- “Troubleshooting” on page 20-15
- “About ToolchainInfo” on page 20-16

Adding a Custom Toolchain

In this section...

“Introduction” on page 20-2

“Create a toolchain specification file” on page 20-2

“Create a handle object for ToolchainInfo” on page 20-2

“Populate ToolchainInfo” on page 20-4

“Populate ToolchainInfo’s Build Tools” on page 20-4

“Populate ToolchainInfo’s Build Configurations” on page 20-8

Introduction

This procedure shows how to create a script that defines and registers a new toolchain. When you have completed this process, you can use the toolchain to compile an executable or library

Some examples in this procedure refer to a specific toolchain. However, you can apply the same approach with other toolchains.

Typically, your file contains a function that returns a ToolchainInfo object. Save this object in a MAT file that the software can load and use during code generation.

Create a toolchain specification file

In MATLAB, enter **Ctrl+N** to create a new script.

For the first line of the script, enter: `function tc = my_custom_toolchain()`

Create a handle object for ToolchainInfo

For line 2, enter: `tc = coder.make.ToolchainInfo(name-value pair,name-value pair,)`

The `coder.make.ToolchainInfo()` constructor creates a handle, called `tc` for interacting with `ToolchainInfo`. This handle points to the `ToolchainInfo`, not a copy of `ToolchainInfo`.

Use multiple name-value pairs to specify each of the `ToolchainInfo()` parameter values your toolchain requires.

BuildArtifact Parameter

You can use the `BuildArtifact` parameter to override the default make tool for your operating system. Use this parameter if you intent to use the toolchain definition on a computer with an operating system that is different from the one you are currently using.

If you do not provide a name value pair for this parameter, the default values are:

- `'nmake makefile'` for Microsoft Windows.
- `'gmake makefile'` for Linux and Mac.

For example:`tc = coder.make.ToolchainInfo('BuildArtifact', 'gmake makefile');`

SupportedLanguages Parameter

You can use the `SupportedLanguages` parameter to override the default programming language, `'C/C++'`, and to initialize `ToolchainInfo` with the default build tools for the specified language(s).

For example: `tc = coder.make.ToolchainInfo('SupportedLanguages', 'C');`

You can set `SupportedLanguages` to any of the following values:

- `'C'`
- `'C++'`
- `'C/C++'` (the default value)
- `'Asm/C'`

- 'Asm/C++'
- 'Asm/C/C++'

Populate ToolchainInfo

Assign values to ToolchainInfo properties and attributes. For more information see “Reference”.

For example:

```
GCC:
tc.Name           = 'GNU gcc/g++ v4.4.x | gmake (64-bit Linux)';
tc.Platform       = 'glnxa64'; %you create a separate toolchain for each
tc.SupportedVersion = '4.4'; %Metadata used for information.
tc.Revision       = '1.0'; %can be used for your tracking.
```

```
%Specify the toolchain's inlined commands:
%Commands are put verbatim in Makefile.
tc.InlinedCommands = '!include <ntwin32.mak>';
```

```
%Specify the toolchain's attributes:
GCC:
tc.addAttribute('SupportsSpacesInPaths', false);
tc.addAttribute('SupportsUNCPaths', false);
tc.addAttribute('SupportsDoubleQuotes', false);
```

```
%Specify the toolchain's setup and cleanup commands:
MSVC:
tc.MATLABSetup{1} = 'mex('-setup:mssdk71', '-f', 'setup_mssdk71.bat')';
tc.ShellSetup{1}  = 'call "setup_mssdk71.bat"';
tc.MATLABCleanup = 'delete(''setup_mssdk71.bat'')';
tc.ShellCleanup  = [];
```

```
%Specify macros that the toolchain will use:
tc.addMacro('MW_BIN_DIR', '$(MATLAB_ROOT)/bin/win64');
```

Populate ToolchainInfo's Build Tools

You can specify attributes for any of the following properties for ToolchainInfo.

Property	Data Type	Brief Description
PrebuildTools	<code>coders.make.util.OrderedSet</code> of <code>coders.make.BuildTool</code> 's	The list of tools that run before the compilation stage. Default: <empty list>
BuildTools	<code>coders.make.util.OrderedSet</code> of <code>coders.make.BuildTool</code> 's	The list of tools used for compilation, archiving and linking. Default: <ul style="list-style-type: none"> • Assembler* • C Compiler • C++ Linker* • Archiver • Linker *depends on the supported language
PostbuildTools	<code>coders.make.util.OrderedSet</code> of <code>coders.make.BuildTool</code> 's	The list of tools that run after the build stage. Default: <ul style="list-style-type: none"> • Download <empty> • Execute <empty>
BuilderApplication	<code>coders.make.BuildTool</code>	The tool used in putting together all build tools to generate the final binaries (e.g. gmake) Default: depends on the constructor parameters

Specify attributes of each build tool

For example, specify the attributes of the BuildTools property”

```
% -----  
% C Compiler  
% -----  
  
tool = tc.getBuildTool('C Compiler');  
tool.setName(          'GNU C Compiler');  
tool.setCommand(       'gcc');  
tool.setPath(          '');  
  
tool.setDirective(     'IncludeSearchPath', '-I');  
tool.setDirective(     'PreprocessorDefine', '-D');  
tool.setDirective(     'OutputFlag',       '-o');  
tool.setDirective(     'Debug',            '-g');  
  
tool.setFileExtension( 'Source',  '.c');  
tool.setFileExtension( 'Header',  '.h');  
tool.setFileExtension( 'Object',  '.o');  
  
% -----  
% C++ Compiler  
% -----  
  
tool = tc.getBuildTool('C++ Compiler');  
  
tool.setName(          'GNU C++ Compiler');  
tool.setCommand(       'g++');  
tool.setPath(          '');  
  
tool.setDirective(     'IncludeSearchPath', '-I');  
tool.setDirective(     'PreprocessorDefine', '-D');  
tool.setDirective(     'OutputFlag',       '-o');  
tool.setDirective(     'Debug',            '-g');  
  
tool.setFileExtension( 'Source',  '.cpp');  
tool.setFileExtension( 'Header',  '.hpp');  
tool.setFileExtension( 'Object',  '.o');  
  
% -----  
% Linker  
% -----
```



```

tool = tc.getBuildTool('Linker');

tool.setName(          'GNU Linker');
tool.setCommand(      'gcc');
tool.setPath(         '');

tool.setDirective(    'Library',          '-l');
tool.setDirective(    'LibrarySearchPath', '-L');
tool.setDirective(    'OutputFlag',       '-o');
tool.setDirective(    'Debug',            '-g');

tool.setFileExtension('Executable',      '');
tool.setFileExtension('Shared Library',  '.so');

% -----
% Archiver
% -----

tool = tc.getBuildTool('Archiver');

tool.setName(          'GNU Archiver');
tool.setCommand(      'ar');
tool.setPath(         '');

tool.setDirective(    'OutputFlag',       '');

tool.setFileExtension('Static Library',  '.a');

```

Specify the platform the where the BuilderApplication will be used

```
tc.setBuilderApplication(platform);
```

The BuilderApplication is already initialized to either gmake to nmake tool during construction. The BuildArtifact property dictates which builder application tool is going to be used. The line above calibrates the BuilderApplication to be suitable for the platform with which it will be used.

Modify the CommandPattern if the default does not apply

MSVC:

```
tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|
```

Populate ToolchainInfo's Build Configurations

For each build configuration, provide the toolchain options/flags.

```
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption( 'C Compiler',           '-c $(ANSI_OPTS) -O0');
cfg.setOption( 'C++ Compiler',        '-c $(CPP_ANSI_OPTS) -O0');
cfg.setOption( 'Linker',               '-Wl,-rpath,"$(MW_BIN_DIR)"');
cfg.setOption( 'Shared Library Linker', '-shared
                                     -Wl,-rpath,"$(MW_BIN_DIR)",-L"$(MW_BIN_DIR)"');
cfg.setOption( 'Archiver',             'ruvs');
```

```
cfg = tc.getBuildConfiguration('Faster Runs');
cfg.setOption( 'C Compiler',           '-c $(ANSI_OPTS) -O3 -fno-lto');
cfg.setOption( 'C++ Compiler',        '-c $(CPP_ANSI_OPTS) -O3 -fno-lto');
cfg.setOption( 'Linker',               '-Wl,-rpath,"$(MW_BIN_DIR)"');
cfg.setOption( 'Shared Library Linker', '-shared
                                     -Wl,-rpath,"$(MW_BIN_DIR)",-L"$(MW_BIN_DIR)"');
cfg.setOption( 'Archiver',             'ruvs');
```

```
cfg = tc.getBuildConfiguration('Debug');
cfg.setOption( 'C Compiler',           '-c $(ANSI_OPTS) -O0 -g');
cfg.setOption( 'C++ Compiler',        '-c $(CPP_ANSI_OPTS) -O0 -g');
cfg.setOption( 'Linker',               '-Wl,-rpath,"$(MW_BIN_DIR)"');
cfg.setOption( 'Shared Library Linker', '-shared
                                     -Wl,-rpath,"$(MW_BIN_DIR)",-L"$(MW_BIN_DIR)"');
cfg.setOption( 'Archiver',             'ruvs');
```

If the options are the same across all build configurations, you can use the following syntax:

```
tc.setBuildConfigurationOption('all', 'Download',  '');
tc.setBuildConfigurationOption('all', 'Execute',   '');
tc.setBuildConfigurationOption('all', 'Make Tool', '-f $(MAKEFILE)');
```

Register a Toolchain

In this section...

“Create a ToolchainInfo MAT file” on page 20-9

“Define toolchain in rtwTargetInfo.m” on page 20-9

“Provide info about the toolchain” on page 20-10

“Reset the toolchain registry” on page 20-10

Create a ToolchainInfo MAT file

From the ToolchainInfo MATLAB file, generate the populated ToolchainInfo object. Save this to a variable such as tc, then save this variable to a MAT file.

Define toolchain in rtwTargetInfo.m

Create or use an existing rtwTargetInfo.m to register the toolchain.

Creating an rtwTargetInfo.m file

Create a file named rtwTargetInfo.m and provide the following code. Make sure this is in the MATLAB path.

```
function rtwTargetInfo(tr)
    tr.registerTargetInfo(@loc_createToolchain);
end
function config = loc_createToolchain
    config = coder.make.ToolchainInfoRegistry; % initialize
end
```

Using an rtwTargetInfo.m file

Locate an rtwTargetInfo.m file that is in the MATLAB path.

Provide info about the toolchain

Inside `loc_createToolchain` subfunction, add another element to the return argument like so:

```
config(end+1)           = coder.make.ToolchainInfoRegistry;  
config(end).Name       = <Name of toolchain>;  
config(end).Alias      = 'Microsoft-9.0'; % internal use only  
config(end).FileName   = path + MAT filename;  
config(end).TargetHWDeviceType = {'*'};  
config(end).Platform   = {'win64'};
```

To add toolchains in the future, reuse and add to this `rtwTargetInfo.m` file.

Reset the toolchain registry

For the above changes to take effect, reset the registry. Invoke the following on the MATLAB command prompt:

```
RTW.TargetRegistry.getInstance('reset')
```

Determine if a Toolchain is Registered

you

Command Line Approach

A toolchain is registered if you find it in the output of the following commands:

```
>tr = RTW.TargetRegistry.getInstance();
>toolchains = tr.getToolchainInfos();
>toolchains.Name
ans =
    Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)
ans =
    Microsoft Visual C++ 2008 v9.0 | nmake (64-bit Windows)
ans =
    Microsoft Windows SDK v7.1 | nmake (64-bit Windows)
```

Using a Custom Toolchain – Command Line Approach

Create the config object that uses the custom toolchain.

```
cfg = coder.config('exe');  
cfg.CustomSource = 'coderrand_main.c';  
cfg.CustomInclude = pwd;
```

Tell the config object, `cfg`, to use the registered toolchain:

```
cfg.Toolchain = 'Intel v12.1';
```

If you do not have the Intel compilers installed, you can use the following command to generate the code and makefile only.

```
cfg.GenCodeOnly = true;
```

Run the codegen to generate the code and makefile that uses the new toolchain.

```
codegen -config cfg coderrand
```

Once the codegen is finished, and you had Intel compilers installed, you can use

```
'system('coderrand.exe')'
```

to run the executable.

Using a Custom Toolchain – UI Approach

1 Task: Switching toolchains Build your MATLAB Coder files using a different toolchain.

Given: `cfg = coder.config('exe');`

Command Line approach: Set the Toolchain property `cfg.Toolchain = 'Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)';`

GUI approach: Select from the Toolchain pulldown

1. Open the MATLAB Coder's Project Settings dialog. open `cfg`
2. Go to the Toolchain panel.
3. Select a new entry from the Toolchain dropdown.

Removing a Toolchain

You can reset the TargetRegistry to remove the toolchain that you registered above. `RTW.TargetRegistry.getInstance('reset');`

Troubleshooting

About ToolchainInfo

In this section...
“What is a Toolchain?” on page 20-16
“What is ToolchainInfo?” on page 20-16

What is a Toolchain?

In software, a toolchain is the set of programming tools that are used to create a product (typically another computer program or system of programs). The tools may be used in a chain, so that the output of each tool becomes the input for the next, but the term is used widely to refer to any set of linked development tools. A simple software development toolchain consists of a text editor for editing source code, a compiler, an assembler and a linker to transform the source code into an executable program, libraries to provide interfaces to the operating system, and a debugger.

What is ToolchainInfo?

ToolchainInfo is a MATLAB data class that you can use to represent the different components of your toolchain.

To register a new build tool, you define its properties in ToolchainInfo.

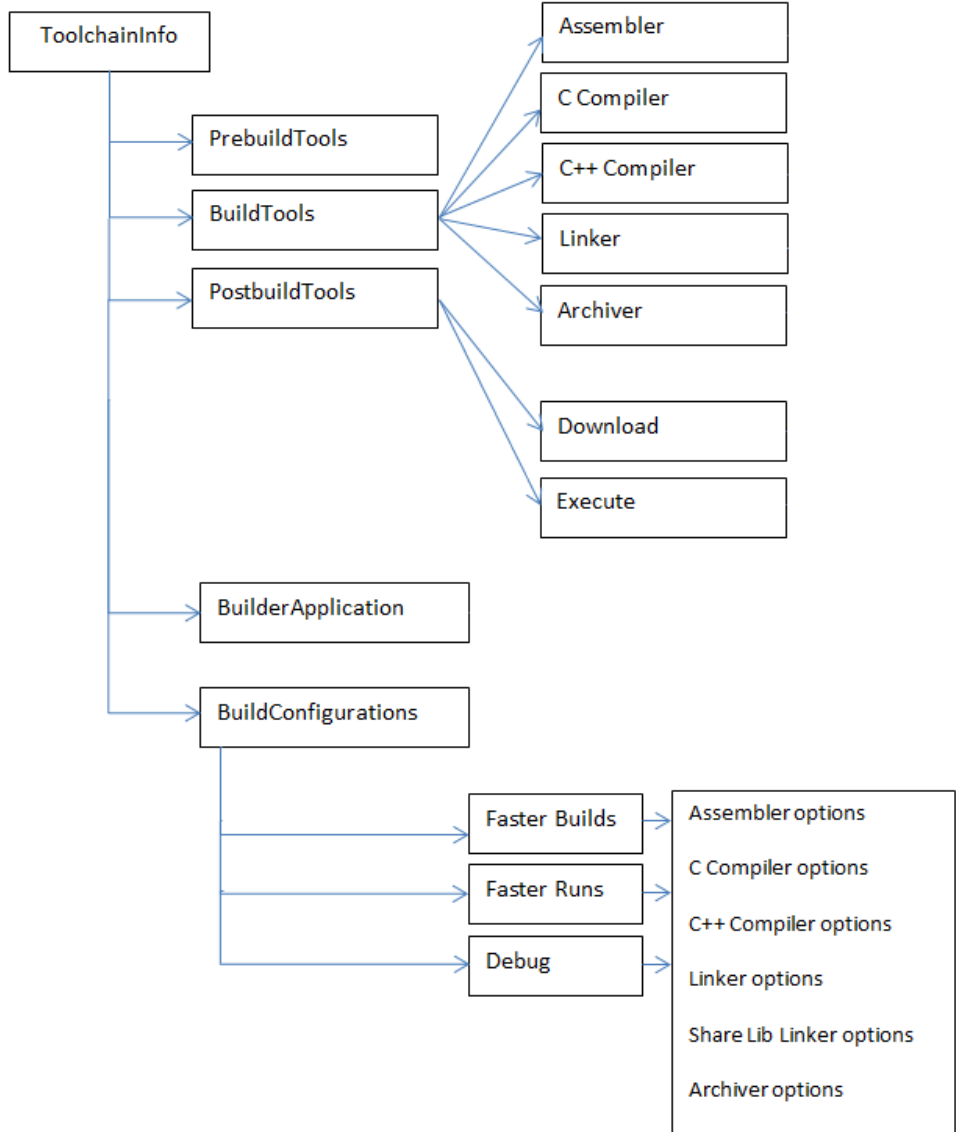
Three properties in ToolchainInfo represent build tools:

- PrebuildTools – Tools used before compiling the source files into object files.
- BuildTools – Tools used for compiling source files and linking/archiving them to form a binary.
- PostbuildTools – Tools used after the linker/archiver is invoked.

If you instantiate ToolchainInfo to support building sources that involve assembler, C, or C++ files, the ToolchainInfo object contains the default set of build tools shown here.

ToolchainInfo class & key properties

Default build tools and options



Deploying Generated Code

- “Call a C Static Library Function from C Code” on page 21-2
- “Call a C/C++ Static Library Function from MATLAB Code” on page 21-4
- “Call Generated C/C++ Functions” on page 21-6
- “Use a MATLAB® Coder™ Dynamic Library in a Simple Microsoft® Visual Studio® Project” on page 21-9
- “Custom C/C++ Code Integration” on page 21-12

Call a C Static Library Function from C Code

This example shows how to call a generated C library function from C code. It uses the C static library function `absval` described in “Call a C/C++ Static Library Function from MATLAB Code” on page 21-4.

- 1 Write a main function in C that does the following:
 - Includes the generated header file, which contains the function prototypes for the library function.
 - Calls the initialize function before calling the library function for the first time.
 - Calls the terminate function after calling the library function for the last time.

Here is an example of a C main function that calls the library function `absval`:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "absval.h"

int main(int argc, char *argv[])
{
    absval_initialize();

    printf("absval(-2.75)=%g\n", absval(-2.75));

    absval_terminate();

    return 0;
}
```

- 2 Configure your target to integrate this custom C main function with your generated code, as described in “Custom C/C++ Code Integration” on page 21-12.

For example, you can define a configuration object that points to the custom C code:

- a** Create a configuration object. At the MATLAB prompt, enter:

```
cfg = coder.config('exe');
```

- b** Set custom code properties on the configuration object, as in these example commands:

```
cfg.CustomSource = 'main.c';  
cfg.CustomInclude = 'c:\myfiles';
```

- 3** Generate the C executable. Use the `-args` option to specify that the input is a real, scalar double. At the MATLAB prompt, enter:

```
codegen -config cfg absval -args {0}
```

- 4** Call the executable. For example:

```
absval(-2.75)
```

Call a C/C++ Static Library Function from MATLAB Code

This example shows how to call a C/C++ library function from MATLAB code that is suitable for code generation.

Suppose you have a MATLAB file `absval.m` that contains the following function:

```
function y = absval(u) %#codegen
    y = abs(u);
end
```

To generate a C static library function and call it from MATLAB code:

1 Generate the C library for `absval.m`.

```
codegen -config:lib absval -args {0,0}
```

Here are key points about this command:

- The `-config:lib` option instructs MATLAB Coder to generate `absval` as a C static library function.

The default target language is C. To change the target language to C++, see “Specify a Language for Code Generation” on page 19-24.

- MATLAB Coder creates the library `absval.lib` (or `absval.a` on Linux Torvalds’ Linux) and header file `absval.h` in the folder `/emcprj/rtwlib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the C library.
- The `-args` option specifies the class, size, and complexity of the primary function input `u` by example, as described in “Define Input Properties by Example at the Command Line” on page 19-43.

2 Write a MATLAB function to call the generated library:

```
%#codegen
function y = callabsval

% Call the initialize function before
% calling the C function for the first time
coder.ceval('absval_initialize');
```



```
y = -2.75;
y = coder.ceval('absval',y);

% Call the terminate function after
% calling the C function for the last time
coder.ceval('absval_terminate');
```

The MATLAB function `callabsval` uses the interface `coder.ceval` to call the generated C functions `absval_initialize`, `absval`, and `absval_terminate`. You must use this function to call C functions from generated code. For more information, see “Call Generated C/C++ Functions” on page 21-6.

- 3** Convert the code in `callabsval.m` to a MEX function so that you can call the C library function `absval` directly from the MATLAB prompt.

- a** Generate the MEX function using `codegen` as follows:

- Create a code generation configuration object for a MEX function:

```
cfg = coder.config
```

- On Microsoft Windows platforms, use this command:

```
codegen -config cfg callabsval codegen/lib/absval/absval.lib
      codegen/lib/absval/absval.h
```

By default, this command creates, in the current folder, a MEX function named `callabsval_mex`

On the Linus Torvalds' Linux platform, use this command:

```
codegen -config cfg callabsval codegen/lib/absval/absval.a
      codegen/lib/absval/absval.h
```

- b** At the MATLAB prompt, call the C library by running the MEX function. For example, on Windows:

```
callabsval_mex
```

Call Generated C/C++ Functions

In this section...

“Conventions for Calling Functions in Generated Code” on page 21-6

“How to Call C/C++ Functions from MATLAB Code” on page 21-6

“Calling Initialize and Terminate Functions” on page 21-7

“Calling C/C++ Functions with Multiple Outputs” on page 21-8

“Calling C/C++ Functions that Return Arrays” on page 21-8

Conventions for Calling Functions in Generated Code

When generating code, MATLAB Coder uses the following calling conventions:

- Passes arrays by reference as inputs.
- Returns arrays by reference as outputs.
- Unless you optimize your code by using the same variable as both input and output, passes scalars by value as inputs. In that case, MATLAB Coder passes the scalar by reference.
- Returns scalars by value for single-output functions.
- Returns scalars by reference:
 - For functions with multiple outputs.
 - When you use the same variable as both input and output.

For more information about optimizing your code by using the same variable as both input and output, see “Eliminate Redundant Copies of Function Inputs (A=foo(A))” on page 19-66.

How to Call C/C++ Functions from MATLAB Code

You can call the C/C++ functions generated for libraries as custom C/C++ code from MATLAB functions that are suitable for code generation. For static libraries, you must use the `coder.ceval` function to wrap the function calls, as in this example:

```
function y = callmyCFunction %#codegen
    y = 1.5;
    y = coder.ceval('myCFunction',y);
end
```

Here, the MATLAB function `callmyCFunction` calls the custom C function `myCFunction`, which takes one input argument.

For dynamically-linked libraries, you can also use `coder.ceval`.

There are additional requirements for calling C/C++ functions from the MATLAB code in the following situations:

- You want to call generated C/C++ libraries or executables from a MATLAB function. Call housekeeping functions generated by MATLAB Coder, as described in “Calling Initialize and Terminate Functions” on page 21-7.
- You want to call C/C++ functions that are generated from MATLAB functions that have more than one output, as described in “Calling C/C++ Functions with Multiple Outputs” on page 21-8.
- You want to call C/C++ functions that are generated from MATLAB functions that return arrays, as described in “Calling C/C++ Functions that Return Arrays” on page 21-8.

Calling Initialize and Terminate Functions

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder automatically generates two housekeeping functions that you must call along with the C/C++ function.

Housekeeping Function	When to Call
<i>primary_function_name_initialize</i>	Before you call your C/C++ executable or library function for the first time
<i>primary_function_name_terminate</i>	After you call your C/C++ executable or library function for the last time

From C/C++ code, you can call these functions directly. However, to call them from MATLAB code that is suitable for code generation, you must use the `coder.ceval` function. `coder.ceval` is a MATLAB Coder function, but is not supported by the native MATLAB language. Therefore, if your MATLAB code uses this function, use `coder.target` to disable these calls in MATLAB and replace them with equivalent functions.

Calling C/C++ Functions with Multiple Outputs

Although MATLAB Coder can generate C/C++ code from MATLAB functions that have multiple outputs, the generated C/C++ code cannot return multiple outputs directly because the C/C++ language does not support multiple return values. Instead, you can achieve the effect of returning multiple outputs from your C/C++ function by using `coder.wref` with `coder.ceval`.

See Also

- “Call Generated C/C++ Functions” on page 21-6
- `coder.wref` function reference information
- `coder.ceval` function reference information

Calling C/C++ Functions that Return Arrays

Although MATLAB Coder can generate C/C++ code from MATLAB functions that return values as arrays, the generated code cannot return arrays *by value* because the C/C++ language is limited to returning single, scalar values. Instead, you can return arrays from your C/C++ function *by reference* as pointers by using `coder.wref` with `coder.ceval`.

See Also

- “Call Generated C/C++ Functions” on page 21-6
- `coder.wref` function reference information
- `coder.ceval` function reference information

Use a MATLAB Coder Dynamic Library in a Simple Microsoft Visual Studio Project

These steps outline how to create and configure a simple Microsoft Visual Studio® Win32 Console Application project to call a dynamic library (DLL) that was generated by MATLAB Coder. This procedure provides information on how to do this in Microsoft Visual Studio 2008, the steps might differ in other versions of Microsoft Visual Studio.

- 1 Create a MATLAB function `foo` and save it as `foo.m` in a local writable folder, for example, `c:\dll_test`.

```
function c = foo(a) %#codegen
    c = sqrt(a);
end
```

- 2 Generate a DLL for the MATLAB function `foo`, using the `-args` option to specify that the input `a` is a real double.

```
codegen -report -config:dll foo -args {0}
```

On Microsoft Windows systems, `codegen` generates a C dynamic library, `foo.dll`, and supporting files, in the default folder, `codegen/dll/foo`.

- 3 In Microsoft Visual Studio, create an empty Win32 Console Application project.
- 4 Verify that the project configuration specifies architecture that matches your computer. By default, MATLAB Coder builds a DLL for the platform that you are working on, but Microsoft Visual Studio builds for Win32.

In Microsoft Visual Studio 2008:

- a Select **Build > Configuration Manager**.
 - b In the **Configuration Manager**, set **Active solution platform** to match your platform.
- 5 Configure the project to use the release version of the C run-time library. By default, the Microsoft Visual Studio project uses the debug version of the C run-time library, but the DLL generated by MATLAB Coder uses the release version. For example, in Microsoft Visual Studio 2008:

- a** Select **Build > Configuration Manager**.
 - b** In the **Configuration Manager**, set **Active solution configuration** to **Release**.
- 6** Create a main file that calls `foo.dll`. The main function **must**:
 - Include the generated header file, which contains the function prototypes for the library function.
 - Call the initialize function before calling the library function for the first time.
 - Call the terminate function after calling the library function for the last time.

For example:

```
#include "foo.h"
#include "foo_initialize.h"
#include "foo_terminate.h"
#include <stdio.h>

int main()
{
    foo_initialize();
    printf("%f\n", foo(25));
    foo_terminate();
    getchar();
    return 0;
}
```

- 7** Add the main file to the project.
- 8** In the project, add the folder containing the generated header file to the list of additional include directories. For example, in Microsoft Visual Studio 2008:
 - a** Right-click the project name and select **Properties**.
 - b** Under **C/C++ > General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Include Directories**.

- 9 Add the folder containing the `.lib` file (by default, this is the folder containing the `.dll`) to the list of additional library directories. For example, in Microsoft Visual Studio 2008:
 - a Right-click the project name and select **Properties**.
 - b Under **Linker > General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Library Directories**.
- 10 Add the `.lib` file name to the list of additional libraries. For example, in Microsoft Visual Studio 2008:
 - a Right-click the project name and select **Properties**.
 - b Under **Linker > Input**, add `foo.lib` to **Additional Dependencies**.

You are now ready to build your project.

Note To run the application, you must either add the folder containing the generated DLL to your path or run from the folder that contains the DLL.

Custom C/C++ Code Integration

In this section...

“About Custom C/C++ Code Integration with MATLAB® Coder™” on page 21-12

“Specifying Custom C/C++ Files in the Project Settings Dialog Box” on page 21-12

“Specifying Custom C/C++ Files at the Command Line” on page 21-13

“Specifying Custom C/C++ Files with Configuration Objects” on page 21-13

About Custom C/C++ Code Integration with MATLAB Coder

You integrate custom C/C++ code with generated C/C++ code by specifying the locations of your external source files, header files, and libraries to MATLAB Coder. You can specify custom C/C++ files from the project settings dialog box, the command line, or with configuration objects.

Specifying Custom C/C++ Files in the Project Settings Dialog Box

- 1 On the project **Build** tab, click the **More settings** link to open the Project Settings dialog box.
- 2 On the **Custom Code** tab, under **Custom C-code to include in generated files**, specify **Source file** and **Header file**. **Source file** specifies the code to appear at the top of generated C/C++ source files. **Header file** specifies the code to appear at the top of generated header files.

Custom Code Property	Description
	Under Additional files and directories to be built , provide an absolute path or a path relative to the project folder.
Include directories	Specifies a list of folders that contain custom header, source, object, or library files. Separate list items with a semicolon.

Custom Code Property	Description
Source files	Specifies additional custom C/C++ files to be compiled with the MATLAB file. Separate list items with a semicolon.
Libraries	Specifies the names of object or library files to be linked with the generated code. Separate list items with a semicolon.
Under Custom C-code to include in generated files	
Source file	Specifies code to appear at the top of generated C/C++ source files.
Header file	Specifies custom code to appear at the top of generated header files

Specifying Custom C/C++ Files at the Command Line

When you compile MATLAB function with MATLAB Coder, you can specify custom C/C++ files — such as source, header, and library files — on the command line along with your MATLAB file. For example, suppose you want to generate an embeddable C code executable that integrates a custom C function `myCfcn` with a MATLAB function `myMfcn` that has no input parameters. The custom source and header files for `myCfcn` reside in the folder `C:\custom`. You can use the following command to generate the code:

```
codegen C:\custom\myCfcn.c C:\custom\myCfcn.h myMfcn
```

Specifying Custom C/C++ Files with Configuration Objects

You can specify custom C/C++ files by setting custom code properties on configuration objects.

- 1 Define a configuration object, as described in “Creating Configuration Objects” on page 19-31.

For example:

```
cc = coder.config('lib');
```

- 2 Set one or more of the custom code properties.

Custom Code Property	Description
CustomInclude	<p>Specifies a list of folders that contain custom header, source, object, or library files.</p> <hr/> <p>Note If your folder path name contains spaces, you must enclose it in double quotes:</p> <pre>cc.CustomInclude = 'C:\Program Files\MATLAB\work'</pre> <hr/>
CustomSource	Specifies additional custom C/C++ files to be compiled with the MATLAB file.
CustomLibrary	Specifies the names of object or library files to be linked with the generated code.
CustomSourceCode	Specifies code to insert at the top of each generated C/C++ source file.
CustomHeaderCode	Specifies custom code to insert at the top of each generated C/C++ header file.

For example:

```
cc.CustomInclude = 'C:\custom\src C:\custom\lib';
cc.CustomSource = 'cfunction.c';
cc.CustomLibrary = 'chelper.obj clibrary.lib';
cc.CustomSourceCode = '#include "cgfunction.h"';
```

- 3 Compile the MATLAB code specifying the code generation configuration object.

Note If you generate code for a function that has input parameters, you must specify the inputs. For more information, see “Primary Function Input Specification” on page 19-38.

```
codegen -config cc myFunc
```

4 Call custom C/C++ functions.

From...	Call...
C/C++ source code	Custom C/C++ functions directly
MATLAB code, compiled on the MATLAB Coder path	Custom C/C++ functions using <code>coder.ceval</code> .

For example, from MATLAB code:

```
...  
y = 2.5;  
y = coder.ceval('myFunc',y);  
...
```

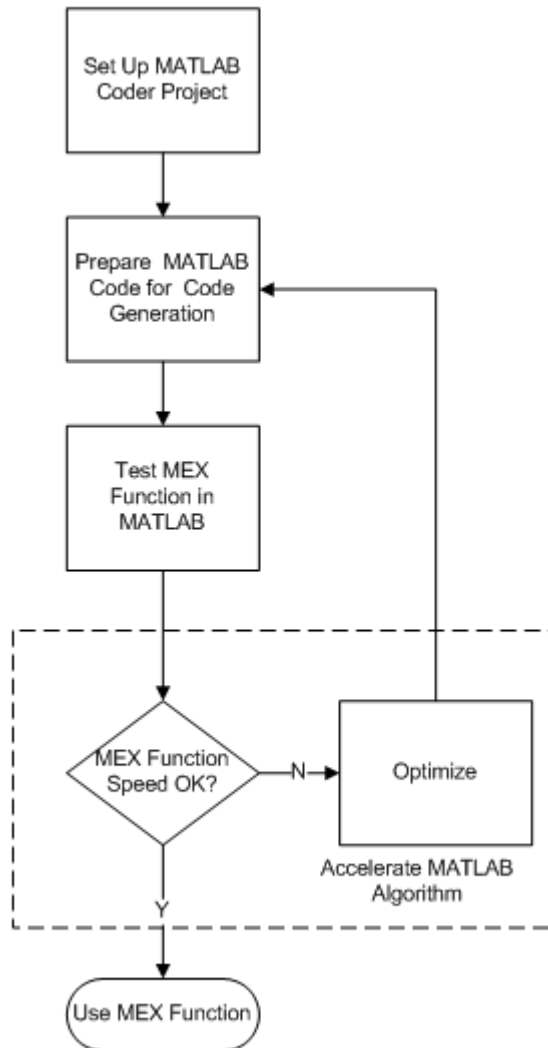
See Also

- “Call Generated C/C++ Functions” on page 21-6

Accelerating MATLAB Algorithms

- “Workflow for Accelerating MATLAB Algorithms” on page 22-2
- “Edge Detection on Images” on page 22-4
- “Accelerate MATLAB Algorithms” on page 22-11
- “Modifying MATLAB Code for Acceleration” on page 22-12
- “Control Run-Time Checks” on page 22-18
- “Acceleration of MATLAB Algorithms Using Parallel for-loops (parfor)” on page 22-21
- “Reduction Assignments in parfor-loops” on page 22-28
- “Classification of Variables in parfor-loops” on page 22-29
- “Accelerate MATLAB Algorithms That Use Parallel for-loops (parfor)” on page 22-40
- “Accelerate MATLAB Algorithms That Use Parallel for-loops (parfor) Specifying the Maximum Number of Threads” on page 22-41
- “Troubleshooting parfor-loops” on page 22-42
- “Accelerating Simulation of Bouncing Balls” on page 22-44

Workflow for Accelerating MATLAB Algorithms



See Also

- “MATLAB® Coder™ Project Set Up Workflow” on page 16-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “Modifying MATLAB Code for Acceleration” on page 22-12

Edge Detection on Images

This example shows how to generate a standalone C library from MATLAB code that implements a simple Sobel filter that performs edge detection on images. The example also shows how to generate and test a MEX function in MATLAB prior to generating C code to verify that the MATLAB code is suitable for code generation.

Prerequisites

To run this example, you must install a C compiler and set it up using the 'mex -setup' command. For more information, see [Setting Up Your C Compiler](#).

Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will only contain the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), you should change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_edge_detection');
```

About the 'sobel' Function

The `sobel.m` function takes an image (represented as a double matrix) and a threshold value and returns an image with the edges detected (based on the threshold value).

```
type sobel
```

```
% edgeImage = sobel(originalImage, threshold)
% Sobel edge detection. Given a normalized image (with double values)
% return an image where the edges are detected w.r.t. threshold value.
function edgeImage = sobel(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));
```



```
k = [1 2 1; 0 0 0; -1 -2 -1];
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k, 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Generate the MEX Function

Generate a MEX function using the 'codegen' command.

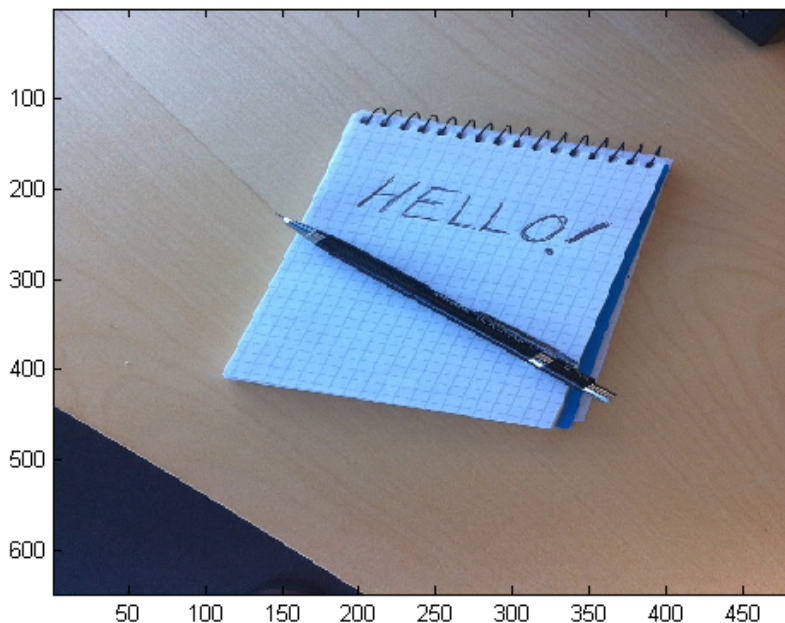
```
codegen sobel
```

Before generating C code, you should first test the MEX function in MATLAB to ensure that it is functionally equivalent to the original MATLAB code and that no run-time errors occur. By default, 'codegen' generates a MEX function named 'sobel_mex' in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Read in the Original Image

Use the standard 'imread' command.

```
im = imread('hello.jpg');
image(im);
```



Convert Image to a Grayscale Version

Convert the color image (shown above) to an equivalent grayscale image with normalized values (0.0 for black, 1.0 for white).

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 *
```

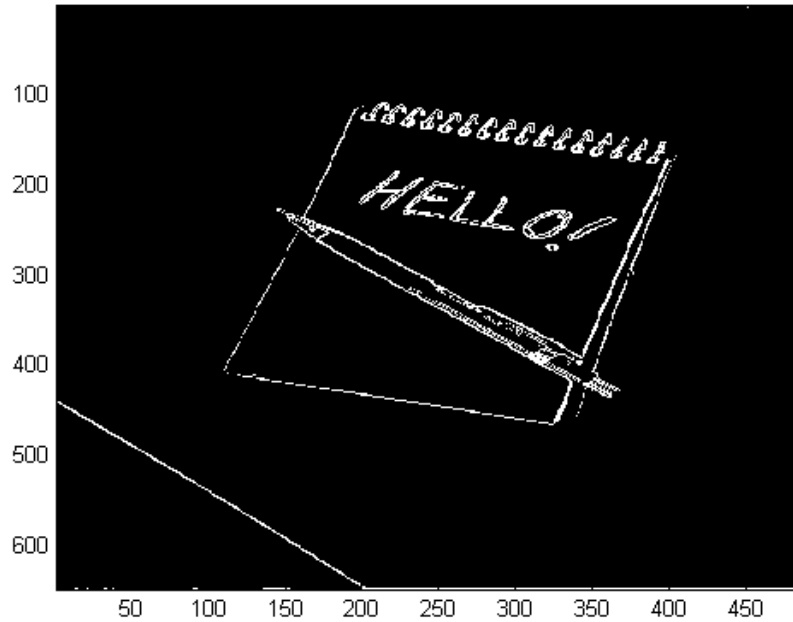
Run the MEX Function (The Sobel Filter)

Pass the normalized image and a threshold value.

```
edgeIm = sobel_mex(gray, 0.7);
```

Display the Result

```
im3 = repmat(edgeIm, [1 1 3]);  
image(im3);
```



Generate Standalone C Code

```
codegen -config coder.config('lib') sobel
```

Using 'codegen' with the '-config coder.config('lib')' option produces a standalone C library. By default, the code generated for the library is in the folder `codegen/lib/sobel/`

Inspect the Generated Function

```
type codegen/lib/sobel/sobel.c
```

```
/*
 * sobel.c
 *
```

```
* Code generation for function 'sobel'
*
* C source code generated on: Thu Jan 17 12:32:59 2013
*
*/

/* Include files */
#include "rt_nonfinite.h"
#include "sobel.h"
#include "sobel_emxutil.h"
#include "sqrt.h"
#include "conv2.h"

/* Function Declarations */
static real_T rt_roundd_snf(real_T u);

/* Function Definitions */
static real_T rt_roundd_snf(real_T u)
{
    real_T y;
    if (fabs(u) < 4.503599627370496E+15) {
        if (u >= 0.5) {
            y = floor(u + 0.5);
        } else if (u > -0.5) {
            y = -0.0;
        } else {
            y = ceil(u - 0.5);
        }
    } else {
        y = u;
    }

    return y;
}

void sobel(const emxArray_real_T *originalImage, real_T threshold,
           emxArray_uint8_T *edgeImage)
{
    emxArray_real_T *H;
    emxArray_real_T *V;
```

```

int32_T b_H;
int32_T c_H;
emxInit_real_T(&H, 2);
emxInit_real_T(&V, 2);

/* edgeImage = sobel(originalImage, threshold) */
/* Sobel edge detection. Given a normalized image (with double values) *
/* return an image where the edges are detected w.r.t. threshold value.
conv2(originalImage, H);
b_conv2(originalImage, V);
b_H = H->size[0] * H->size[1];
emxEnsureCapacity((emxArray__common *)H, b_H, (int32_T)sizeof(real_T));
b_H = H->size[0];
c_H = H->size[1];
c_H *= b_H;
for (b_H = 0; b_H < c_H; b_H++) {
    H->data[b_H] = H->data[b_H] * H->data[b_H] + V->data[b_H] * V->data[b_H];
}

emxFree_real_T(&V);
b_sqrt(H);
b_H = edgeImage->size[0] * edgeImage->size[1];
edgeImage->size[0] = H->size[0];
edgeImage->size[1] = H->size[1];
emxEnsureCapacity((emxArray__common *)edgeImage, b_H, (int32_T)sizeof(uint8_T));
c_H = H->size[0] * H->size[1];
for (b_H = 0; b_H < c_H; b_H++) {
    edgeImage->data[b_H] = (uint8_T)rt_roundd_snf((real_T)(H->data[b_H] >
        threshold) * 255.0);
}

emxFree_real_T(&H);
}

/* End of code generation (sobel.c) */

```

Cleanup

Remove files and return to original folder

Run Command: Cleanup

```
cleanup
```

Accelerate MATLAB Algorithms

For many applications, you can generate MEX functions to accelerate MATLAB algorithms. If you have a Fixed-Point Designer license, you can generate MEX functions to accelerate fixed-point MATLAB algorithms. After generating a MEX function, test it in MATLAB to verify that its operation is functionally equivalent to the original MATLAB algorithm. Then compare the speed of execution of the MEX function with that of the MATLAB algorithm. If the MEX function speed is not sufficiently fast, you might improve it using one of the following methods:

- Choosing a different C/C++ compiler.

It is important that you use a C/C++ compiler that is designed to generate high performance code.

Note The default MATLAB compiler for Windows 32-bit platforms, `lcc`, is designed to generate code quickly. It is not designed to generate high performance code.

- “Modifying MATLAB Code for Acceleration” on page 22-12
- “Control Run-Time Checks” on page 22-18

Modifying MATLAB Code for Acceleration

In this section...

“How to Modify Your MATLAB Code for Acceleration” on page 22-12

“Unroll for-loops” on page 22-12

“Inline Code” on page 22-14

“Eliminate Redundant Copies of Function Inputs (A=foo(A))” on page 22-15

How to Modify Your MATLAB Code for Acceleration

You might improve the efficiency of the generated code using one of the following optimizations:

- “Unroll for-loops” on page 19-63
- “Inline Code” on page 19-65
- “Eliminate Redundant Copies of Function Inputs (A=foo(A))” on page 19-66

Unroll for-loops

Unrolling for-loops eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant.

You can also force loop unrolling for individual functions by wrapping the loop header in a `coder.unroll` function. For more information, see `coder.unroll`.

Limiting Copying the Body of a for-loop in Generated Code

To limit the number of times to copy the body of a for-loop in generated code:

- 1 Write a MATLAB function `getrand(n)` that uses a for-loop to generate a vector of length `n` and assign random numbers to specific elements. Add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the for-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
```



```

% The directive %#codegen indicates that the function
% is intended for code generation
% Calling getrand 8 times triggers unroll
y1 = getrand(8);
% Calling getrand 50 times does not trigger unroll
y2 = getrand(50);

function y = getrand(n)
% Turn off inlining to make
% generated code easier to read
coder.inline('never');

% Set flag variable dounroll to repeat loop body
% only for fewer than 10 iterations
dounroll = n < 10;
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
for i = coder.unroll(1:2:n, dounroll)
    if (i > 2) && (i < n-2)
        y(i) = rand();
    end;
end;
% Loop body ends

```

- 2** In the default output folder, `codegen/lib/test_unroll`, generate C library code for `test_unroll` :

```
codegen -config:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the for-loop (unrolls the loop) because the number of iterations is less than 10:

```

static void m_getrand(real_T y[8])
{
    int32_T i0;
    for(i0 = 0; i0 < 8; i0++) {
        y[i0] = 0.0;
    }
}

```

```
    /* Loop body begins */  
    y[2] = m_rand();  
    y[4] = m_rand();  
    /* Loop body ends */  
}
```

The generated C code for `getrand(50)` does not unroll the for-loop because the number of iterations is greater than 10:

```
static void m_b_getrand(real_T y[50])  
{  
    int32_T i;  
    for(i = 0; i < 50; i++) {  
        y[i] = 0.0;  
    }  
    /* Loop body begins */  
    for(i = 0; i < 50; i += 2) {  
        if((i + 1 > 2) && (i + 1 < 48)) {  
            y[i] = m_rand();  
        }  
    }  
    /* Loop body ends */  
}
```

Inline Code

MATLAB uses internal heuristics to determine whether or not to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. For more information, see `coder.inline`.

Preventing Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)  
    coder.inline('never');  
    y = x;  
end
```

Using Inlining in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
    coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
    coder.inline('never');
end

if any(divisor == 0)
    error('Can not divide by 0');
end

y = dividend / divisor;
```

Eliminate Redundant Copies of Function Inputs (A=foo(A))

You can reduce the number of copies in your generated code by writing functions that use the same variable as both an input and an output. For example:

```
function A = foo( A, B ) %#codegen
A = A * B;
end
```

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, MATLAB passes the variable by

reference in the generated code instead of redundantly copying the input to a temporary variable. In the preceding example, input A is passed by reference in the generated code because it also acts as an output for function foo:

```
...
/* Function Definitions */
void foo(real_T *A, real_T B)
{
    *A *= B;
}
...
```

The reference parameter optimization reduces memory usage and execution time, especially when the variable passed by reference is a large data structure. To achieve these benefits at the call site, call the function with the same variable as both input and output.

By contrast, suppose you rewrite function foo without the optimization:

```
function y = foo2( A, B ) %#codegen
y = A * B;
end
```

MATLAB generates code that passes the inputs by value and returns the value of the output:

```
...
/* Function Definitions */
real_T foo2(real_T A, real_T B)
{
    return A * B;
}
...
```

In some cases, the output of the function cannot be a modified version of its inputs. If you do not use the inputs later in the function, you can modify your code to operate on the inputs instead of on a copy of the inputs. One method is to create additional return values for the function. For example, consider the code:

```
function y1=foo(u1) %#codegen
```

```
x1=u1+1;  
y1=bar(x1);  
end
```

```
function y2=bar(u2)  
% Since foo does not use x1 later in the function,  
% it would be optimal to do this operation in place  
x2=u2.*2;  
% The change in dimensions in the following code  
% means that it cannot be done in place  
y2=[x2,x2];  
end
```

You can modify this code to eliminate redundant copies. The changes are highlighted in bold.

```
function y1=foo(u1) %#codegen  
    u1=u1+1;  
    [y1, u1]=bar(u1);  
end
```

```
function [y2, u2]=bar(u2)  
    u2=u2.*2;  
% The change in dimensions in the following code  
% still means that it cannot be done in place  
y2=[u2,u2];  
end
```

Control Run-Time Checks

In this section...
“Types of Run-Time Checks” on page 22-18
“When to Disable Run-Time Checks” on page 22-19
“How to Disable Run-Time Checks” on page 22-19

Types of Run-Time Checks

The code generated for your MATLAB functions includes the following run-time checks and external calls to MATLAB functions.

- Memory integrity checks

These checks detect violations of memory integrity in code generated for MATLAB functions and stop execution with a diagnostic message.

Caution These checks are enabled by default. Without memory integrity checks, violations result in unpredictable behavior.

- Responsiveness checks in code generated for MATLAB functions

These checks enable periodic checks for Ctrl+C breaks in code generated for MATLAB functions. Enabling responsiveness checks also enables graphics refreshing.

Caution These checks are enabled by default. Without these checks, the only way to end a long-running execution might be to terminate MATLAB.

- Extrinsic calls to MATLAB functions

Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information about extrinsic functions, see “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12.

When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more generated code and slower MEX function execution than generating code with the checks disabled. Similarly, extrinsic calls are time consuming and increase memory usage and execution time. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster MEX function execution. The following table lists issues to consider when disabling run-time checks and extrinsic calls.

Consider disabling...	Only if...
Memory integrity checks	You have already verified that array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C.
Extrinsic calls	You are using extrinsic calls only for functions that do not affect application results.

How to Disable Run-Time Checks

You can disable run-time checks explicitly from the project settings dialog box, the command line, or a MEX configuration dialog box.

Disabling Run-Time Checks in the Project Settings Dialog Box

- 1 On the MATLAB Coder project **Build** tab, click **More settings**.
- 2 On the **Project Settings** dialog box **Speed** tab, clear **Ensure memory integrity**, **Enable responsiveness to CTRL+C and graphics refreshing** or **Extrinsic calls**, as applicable.

Disabling Run-Time Checks From the Command Line

- 1 In the MATLAB workspace, define the MEX configuration object:

```
mexcfg = coder.config('mex');
```

- 2 At the command line, set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties to `false`, as applicable:

```
mexcfg.IntegrityChecks = false;  
mexcfg.ExtrinsicCalls = false;  
mexcfg.ResponsivenessChecks = false;
```


Acceleration of MATLAB Algorithms Using Parallel for-loops (parfor)

In this section...

“Parallel for-loops (parfor) in MEX Functions” on page 22-21

“When to Use parfor-loops” on page 22-22

“When Not to Use parfor-loops” on page 22-23

“Control Compilation of parfor-loops” on page 22-23

“Supported Compilers” on page 22-24

“parfor-Loop Syntax and Restrictions” on page 22-24

“parfor Limitations” on page 22-25

Parallel for-loops (parfor) in MEX Functions

To potentially accelerate execution of generated code, you can generate MEX functions from MATLAB code that contains parallel for-loops (parfor-loops). A parfor-loop in MATLAB software, like the standard MATLAB for-loop, executes a series of statements (the loop body) over a range of values. For more information, see “How parfor-loops Improve Execution Speed” on page 22-22.

So that the generated MEX functions can run in parallel on multiple cores on a desktop, the MATLAB Coder software uses the Open Multi-Processing (OpenMP) application interface to support shared-memory, multicore code generation. If you want distributed parallelism, see Parallel Computing Toolbox™. By default, MATLAB Coder uses up to as many cores as it finds available. If you specify an upper limit on the number of threads to use, MATLAB Coder uses no more than that number of cores, even if additional cores are available. If you request more threads than the number of available cores, MATLAB Coder uses the maximum number of cores available at the time of the call. If there are fewer iterations than threads, some threads perform no work. For more information, see parfor.

Because the loop body can execute in parallel on multiple threads, the loop body must conform to certain restrictions. If MATLAB Coder software detects loops that do not conform to the parfor specification, it does not generate

code and produces an error. For more information, see “What Is Allowed in a parfor-loop” on page 22-24.

How parfor-loops Improve Execution Speed

A parfor-loop might provide better execution speed than its analogous for-loop because several threads can be computing concurrently on the same loop.

Each execution of the body of a parfor-loop is an iteration. The threads evaluate iterations in arbitrary order and independently of each other. Because each iteration is independent, they do not need to be synchronized. If the number of threads is equal to the number of loop iterations, each thread performs one iteration of the loop. If there are more iterations than threads, some threads perform more than one loop iteration.

For example, a loop of 100 iterations could run on 20 threads, so that simultaneously the threads each execute only five iterations of the loop. You might not get quite 20 times improvement in speed because of parallelization overheads, such as thread creation and deletion. So whether your loop takes a long time to run because it has many iterations or because each iteration takes a long time, in most cases, you can improve your loop speed by using multiple threads. Under certain rare circumstances, the use of parfor might increase execution time.

When to Use parfor-loops

Many Iterations of a Simple Calculation

If you need many loop iterations of a simple calculation, parfor divides the loop iterations into groups so that each thread executes some portion of the total number of iterations.

Loop Iterations Take a Long Time to Execute

When you have loop iterations that take a long time to execute, parfor executes the iterations simultaneously on different threads.

When Not to Use parfor-loops

Loop Iterations Are Dependent

When an iteration in your loop depends on the results of other iterations, you cannot use a parfor-loop. If MATLAB Coder software detects loops that do not conform to the parfor specification, it does not generate code and produces an error.

Reductions are an exception to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order. For more information, see “Reduction Variables” on page 22-32.

Small Number of Simple Calculations

You might not accelerate execution for a small number of calculations due to parallelization overheads such as the time taken for thread creation and deletion.

Results Depend on Order of Evaluation of Loop Parameters

Do not rely on the order of evaluation of the `initval`, `endval`, and `numThreads` parameters.

Control Compilation of parfor-loops

By default, MATLAB Coder generates a MEX function that can run the parfor-loop on multiple threads. To generate a MEX function that treats parfor-loops as for-loops and runs on a single thread, disable parfor by using the codegen function `-O disable:openmp` option.

When to Disable parfor

Disable parfor if you want to:

- Compare the execution times of the serial and parallel versions of the generated MEX function.
- Investigate failures. If the parallel version of the generated MEX function fails, disable parfor and generate a serial version to facilitate debugging.

- Use C compilers that do not support OpenMP.

Supported Compilers

You can generate MEX functions capable of running on multiple threads for MATLAB code that contains `parfor`-loops using supported compilers except the Microsoft Visual Studio SDK, Open Watcom, LCC, and Apple Xcode with Clang. If you use the Microsoft Visual Studio SDK, Open Watcom, LCC, or Apple Xcode with Clang, MATLAB Coder treats the `parfor`-loops as `for`-loops and the generated MEX function runs on a single thread.

For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Before generating code, you must set up the compiler. See “Setting Up the C/C++ Compiler”.

parfor-Loop Syntax and Restrictions

parfor Syntax

Use this syntax for a `parfor`-loop:

```
parfor i = M:N  
parfor (i = M:N)
```

Do not use this syntax:

```
parfor (i=M:K:N)  
parfor i=M:K:N
```

To specify the maximum number of threads to use, use this syntax:

```
parfor (i = 1:N, NumThreads)
```

For more information, see `parfor`.

What Is Allowed in a parfor-loop

Assume that each iteration of a `parfor`-loop is evaluated by a different MATLAB thread. If you have a `for`-loop in which the iterations are completely

independent of each other, this loop is a good candidate for a `parfor`-loop. If one iteration depends on the results of another iteration, these iterations are not independent and cannot be evaluated in parallel. Reduction assignments are an exception to the rule that loop iterations must be independent. For more information, see “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 22-37.

The following examples produce equivalent results, with a `for`-loop on the left, and a `parfor`-loop on the right. Try typing each in your MATLAB Command Window.

```
clear A
for i = 1:8
    A(i) = i;
end
A
```

```
clear A
parfor i = 1:8
    A(i) = i;
end
A
```

Each element of `A` is equal to its index. The `parfor`-loop works because each element depends upon only its iteration of the loop. `for`-loops that only repeat such independent tasks are ideally suited candidates for `parfor`-loops.

In a `parfor`-loop, MATLAB Coder does not support variables that it cannot classify. MATLAB Coder classifies variables inside a `parfor`-loop into one of the categories detailed in “Classification of Variables in `parfor`-loops” on page 22-29.

parfor Limitations

The following limitations apply when generating MEX functions for `parfor`-loops.

Nested parfor-Loops

The body of a `parfor`-loop cannot contain another `parfor`-loop. However, it can call a function that contains another `parfor`-loop.

Break and Return Statements

The body of a `parfor`-loop cannot contain `break` or `return` statements.

Global and Persistent Variables

The body of a `parfor`-loop cannot use `global` or `persistent` variable.

MATLAB Classes

MATLAB Coder software does not support reductions on MATLAB classes.

Reductions on char variables

MATLAB Coder software does not support reductions on `char` variables.

For example, you cannot generate C code for the following MATLAB code:

```
c = char(0);
parfor i=1:10
    c = c + char(1);
end
```

In the `parfor`-loop, MATLAB makes `c` a `double`. For code generation, `c` cannot change type.

Calls to External C Code

MATLAB Coder does not support the use of `coder.ceval` in reductions. For example, you cannot generate code for the following `parfor`-loop:

```
parfor i=1:4
    y=coder.ceval('myCFcn',y,i);
end
```

Instead, write a local function that calls the C code using `coder.ceval` and call this function in the `parfor`-loop. For example:

```
parfor i=1:4
    y = callMyCFcn(y,i);
end
...
function y = callMyCFcn(y,i)
    y = coder.ceval('mCyFcn', y , i);
end
```

Extrinsic Calls

You cannot call extrinsic functions in the body of a `parfor`-loop. Calls to functions that contain extrinsic calls result in a run-time error.

rand Functions

MATLAB Coder software does not support calling the `rand`, `randi`, or `randn` functions in the body of a `parfor`-loop.

Inlining Code

MATLAB Coder does not inline functions into `parfor`-loops, including functions that use `coder.inline('always')`.

Unrolling Code

You cannot use `coder.unroll` in `parfor`-loops.

If a loop is unrolled inside a `parfor`-loop, MATLAB Coder cannot classify the variable. For example, consider the following code.

```
for j=coder.unroll(3:6)
    y(i,j)=y(i,j)+i+j;
end
```

This code is unrolled to:

```
y(i,3)=y(i,3)+i+3;
...
y(i,6)=y(i,6)+i+6;
```

In the unrolled code, MATLAB Coder cannot classify the variable `y` because `y` is indexed in different ways inside the `parfor`-loop.

MATLAB Coder does not support variables that it cannot classify. For more information, see “Classification of Variables in `parfor`-loops” on page 22-29.

varargin/varargout

You cannot use `varargin` or `varargout` in `parfor`-loops.

Reduction Assignments in parfor-loops

What are Reduction Assignments?

Reduction assignments, or *reductions*, are an exception to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the loop iterations together, but is independent of the iteration order. For a list of supported reduction variables see “Reduction Variables” on page 22-32.

Multiple Reductions in a parfor-loop

You can perform the same reduction assignment multiple times within a parfor-loop provided that you use the same data type each time.

For example, in the following parfor-loop, $u(i)$ and $v(i)$ must be the same type.

```
parfor i = 1:10;
    X = X + u(i);
    X = X + v(i);
end
```

Similarly, the following example is valid provided that $u(i)$ and $v(i)$ are the same type.

```
parfor i=1:10
    r = foo(r,u(i));
    r = foo(r,v(i));
end
```


Classification of Variables in parfor-loops

In this section...
“Overview” on page 22-29
“Sliced Variables” on page 22-30
“Broadcast Variables” on page 22-32
“Reduction Variables” on page 22-32
“Temporary Variables” on page 22-38

Overview

MATLAB Coder classifies variables inside a `parfor`-loop into one of the categories in the following table. It does not support variables that it cannot classify. If a `parfor`-loop contains variables that cannot be uniquely categorized or if a variable violates its category restrictions, the `parfor`-loop generates an error.

Classification	Description
Loop	Serves as a loop index for arrays
Sliced	An array whose segments are operated on by different iterations of the loop
Broadcast	A variable defined before the loop whose value is used inside the loop, but not assigned inside the loop
Reduction	Accumulates a value across iterations of the loop, regardless of iteration order
Temporary	A variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop

Each of these variable classifications appears in this code fragment:

```
a=0;
c=pi;
z=0;
r=rand(1,10);
parfor i=1:10
```

```

a=i;    % 'a' is a temporary variable
z=z+i;  % 'z' is a reduction variable
b(i)=r(i); % 'b' is a sliced output variable;
        % 'r' a sliced input variable
if i<=c % 'c' is a broadcast variable
    d=2*a; % 'd' is a temporary variable
end
end
end

```

Sliced Variables

A *sliced variable* is one whose value can be broken up into segments, or *slices*, which are then operated on separately by different threads. Each iteration of the loop works on a different slice of the array.

In the next example, a slice of A consists of a single element of that array:

```

parfor i = 1:length(A)
    B(i) = f(A(i));
end

```

Characteristics of a Sliced Variable

A variable in a parfor-loop is sliced if it has the following characteristics:

- **Type of First-Level Indexing** — The first level of indexing is parentheses, ().
- **Fixed Index Listing** — Within the first-level parenthesis, the list of indices is the same for all occurrences of a given variable.
- **Form of Indexing** — Within the list of indices for the variable, exactly one index involves the loop variable.
- **Shape of Array** — In assigning to a sliced variable, the right-hand side of the assignment is not [] or '' (these operators indicate deletion of elements).

Type of First-Level Indexing. For a sliced variable, the first level of indexing is enclosed in parentheses, (). For example, A(...). If you reference a variable using dot notation, A.x, the variable is not sliced.

Variable A on the left is not sliced; variable A on the right is sliced:

`A.q(i,12)`

`A(i,12).q`

Fixed Index Listing. Within the first-level parentheses of a sliced variable's indexing, the list of indices is the same for all occurrences of a given variable.

Variable `B` on the left is not sliced because `B` is indexed by `i` and `i+1` in different places. Variable `B` on the right is sliced.

```
parfor i = 1:10
    B(i) = B(i+1) + 1;
end
```

```
parfor i = 1:10
    B(i+1) = B(i+1) + 1;
end
```

Form of Indexing. Within the list of indices for a sliced variable, one index is of the form `i`, `i+k`, `i-k`, `k+i`, or `k-i`.

- `i` is the loop variable.
- `k` is a constant or a simple (nonindexed) variable.
- Every other index is a constant, a simple variable, colon, or end.

When you use other variables along with the loop variable to index an array, you cannot set these variables inside the loop. These variables are constant over the execution of the entire `parfor` statement. You cannot combine the loop variable with itself to form an index expression.

In the following examples, `i` is the loop variable, `j` and `k` are nonindexed variables.

Variable A Is Not Sliced	Variable A Is Sliced
<code>A(i+f(k),j,:,3)</code>	<code>A(i+k,j,:,3)</code>
<code>A(i,20:30,end)</code>	<code>A(i,:,end)</code>
<code>A(i,:,s.field1)</code>	<code>A(i,:,k)</code>

Shape of Array. A sliced variable must maintain a constant shape. In the following examples, the variable `A` is not sliced:

```
A(i,:) = [];
A(end + 1) = i;
```

Broadcast Variables

A *broadcast variable* is a variable other than the loop variable or a sliced variable that is not modified inside the loop.

Reduction Variables

A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order.

This example shows a `parfor`-loop that uses a scalar reduction assignment. It uses the reduction variable `x` to accumulate a sum across 10 iterations of the loop. The execution order of the iterations on the threads does not matter.

```
x = 0;
parfor i = 1:10
    x = x + i;
end
x
```

Where `expr` is a MATLAB expression, reduction variables appear on both sides of an assignment statement.

<code>X = X + expr</code>	<code>X = expr + X</code>
<code>X = X - expr</code>	See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 22-37
<code>X = X .* expr</code>	<code>X = expr .* X</code>
<code>X = X * expr</code>	<code>X = expr * X</code>
<code>X = X & expr</code>	<code>X = expr & X</code>
<code>X = X expr</code>	<code>X = expr X</code>
<code>X = min(X, expr)</code>	<code>X = min(expr, X)</code>

<code>X = max(X, expr)</code>	<code>X = max(expr, X)</code>
<code>X=f(X, expr)</code> Function f must be a user-defined function.	<code>X = f(expr, X)</code> See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 22-37

Each of the allowed statements is referred to as a *reduction assignment*. A reduction variable can appear only in assignments of this type.

The following example shows a typical usage of a reduction variable X:

```
X = ...;           % Do some initialization of X
parfor i = 1:n
    X = X + d(i);
end
```

This loop is equivalent to the following, where each `d(i)` is calculated by a different iteration:

$$X = X + d(1) + \dots + d(n)$$

If the loop were a regular `for`-loop, the variable X in each iteration would get its value either before entering the loop or from the previous iteration of the loop. However, this concept does not apply to `parfor`-loops.

In a `parfor`-loop, the value of X is not updated directly inside each thread. Rather, additions of `d(i)` are done in each thread, with `i` ranging over the subset of `1:n` being performed on that thread. The software then accumulates the results into X.

Similarly, the reduction:

```
r=r<op> x(i)
```

is equivalent to:

$$r=r<op>x(1)] <op>x(2) \dots <op>x(n)$$

The operation `<op>` is first applied to `x(1) . . . x(n)`, then the partial result is applied to `r`.

If operation <op> takes two inputs, it should meet one of the following criteria:

- Take two arguments of `typeof(x(i))` and return `typeof(x(i))`
- Take one argument of `typeof(r)` and one of `typeof(x(i))` and return `typeof(r)`

Rules for Reduction Variables

Use the same reduction function or operation in all reduction assignments. For a reduction variable, you must use the same reduction function or operation in all reduction assignments for that variable. In the following example, the `parfor`-loop on the left is not valid because the reduction assignment uses `+` in one instance, and `*` in another.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
<pre>parfor i = 1:n if A > 5*k A = A + 1; else A = A * 2; end</pre>	<pre>parfor i = 1:n if A > 5*k A = A * 3; else A = A * 2; end</pre>

Restrictions on reduction function parameter and return types. A reduction `r=r<op> x(i)`, should take arguments of `typeof(x(i))` and return `typeof(x(i))` or take arguments of `typeof(r)` and `typeof(x(i))` and return `typeof(r)`.

In the following example, in the invalid loop, `r` is a fixed-point type and `2` is not. To fix this issue, cast `2` to be the same type as `r`.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
<pre>function r = fiops(in) r=fi(in, 'WordLength',20,... 'FractionLength',14,... 'SumMode', 'SpecifyPrecision',... 'SumWordLength',20,... 'SumFractionLength',14,... 'ProductMode', 'SpecifyPrecision',... 'ProductWordLength',20,... 'ProductFractionLength',14); parfor i = 1:10 r = r*2; end</pre>	<pre>r=fi(in, 'WordLength',20,... 'FractionLength',14,... 'SumMode', 'SpecifyPrecision',... 'SumWordLength',20,... 'SumFractionLength',14,... 'ProductMode', 'SpecifyPrecision',... 'ProductWordLength',20,... 'ProductFractionLength',14); T = r.numericity; F = r.fimath; parfor i = 1:10 r = r*fi(2,T,F); end</pre>

In the following example, the reduction function `fcn` is invalid because it does not handle the case when input `u` is fixed point. (The `+` and `*` operations are automatically polymorphic.) You must write a polymorphic version of `fcn` to handle the expected input types.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
<pre>function [y0, y1, y2] = pfuserfcn(u) y0 = 0; y1 = 1; [F, N] = fiprops(); y2 = fi(1,N,F); parfor (i=1:numel(u),12) y0 = y0 + u(i); y1 = y1 * u(i); y2 = fcn(y2, u(i)); end end function y = fcn(u, v) y = u * v; end</pre>	<pre>function [y0, y1, y2] = pfuserfcn(u) y0 = 0; y1 = 1; [F, N] = fiprops(); y2 = fi(1,N,F); parfor (i=1:numel(u),12) y0 = y0 + u(i); y1 = y1 * u(i); y2 = fcn(y2, u(i)); end end % fcn handles inputs of type double % and fi function y = fcn(u, v) if isa(u,'double') y = u * v; else [F, N] = fiprops(); y = u * fi(v,N,F); end end function [F, N] = fiprops() N = numericity(1,96,30); F = fimath('ProductMode',... 'SpecifyPrecision',... 'ProductWordLength',96); end</pre>

Reduction Assignments, Associativity, and Commutativity of Reduction Functions

Reduction Assignments. MATLAB Coder does not allow reduction variables to be read anywhere in the parfor-loop except in reduction statements. In the following example, the call `foo(r)` after the reduction statement `r=r+i` causes the loop to be invalid.

```
function r = temp %#codegen
    r = 0;
    parfor i=1:10
        r = r + i;
        foo(r);
    end
end
```

Associativity in Reduction Assignments. If you use a user-defined function `f` in the definition of a reduction variable, to get deterministic behavior of parfor-loops, the reduction function `f` must be associative.

Note If `f` is not associative, MATLAB Coder does not generate an error. You must write code that meets this recommendation.

To be associative, the function `f` must satisfy the following for all `a`, `b`, and `c`:

$$f(a, f(b, c)) = f(f(a, b), c)$$

Commutativity in Reduction Assignments. Some associative functions, including `+`, `.`, `min`, and `max`, are also commutative. That is, they satisfy the following for all `a` and `b`:

$$f(a, b) = f(b, a)$$

The function `f` of a reduction assignment must be commutative. If `f` is not commutative, different executions of the loop might result in different answers.

Unless `f` is a known noncommutative built-in, the software assumes that it is commutative.

Temporary Variables

A *temporary variable* is a variable that is the target of a direct, nonindexed assignment, but is not a reduction variable. In the following `parfor`-loop, `a` and `d` are temporary variables:

```
a = 0;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;           % Variable a is temporary
    z = z + i;
    if i <= 5
        d = 2*a;    % Variable d is temporary
    end
end
```

In contrast to the behavior of a `for`-loop, before each iteration of a `parfor`-loop, MATLAB Coder effectively clears temporary variables. Because the iterations must be independent, the values of temporary variables cannot be passed from one iteration of the loop to another. Therefore, temporary variables must be set inside the body of a `parfor`-loop, so that their values are defined separately for each iteration.

A temporary variable in the context of the `parfor` statement is different from a variable with the same name that exists outside the loop.

Uninitialized Temporaries

Because temporary variables are cleared at the beginning of every iteration, MATLAB Coder can detect certain cases in which an iteration through the loop uses the temporary variable before it is set in that iteration. In this case, MATLAB Coder issues a static error rather than a run-time error, because there is little point in allowing execution to proceed if a run-time error will occur. For example, suppose you write:

```
b = true;
parfor i = 1:n
    if b && some_condition(i)
        do_something(i);
        b = false;
    end
end
```

```
    end  
    ...  
end
```

This loop is acceptable as an ordinary `for`-loop, but as a `parfor`-loop, `b` is a temporary variable because it occurs directly as the target of an assignment inside the loop. Therefore, it is cleared at the start of each iteration, so its use in the condition of the `if` is uninitialized. (If you change `parfor` to `for`, the value of `b` assumes sequential execution of the loop, so that `do_something(i)` is executed for only the lower values of `i` until `b` is set `false`.)

Accelerate MATLAB Algorithms That Use Parallel for-loops (parfor)

This example shows how to generate a MEX function for a MATLAB algorithm that contains a parfor-loop.

- 1 Write a MATLAB function that contains a parfor-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
    a(i,:)=real(fft(r(i,:)));
end
```

- 2 Generate a MEX function for test_parfor. At the MATLAB command line, enter:

```
codegen test_parfor
```

codegen generates a MEX function, test_parfor_mex, in the current folder.

- 3 Run the MEX function. At the MATLAB command line, enter:

```
test_parfor_mex
```

Because you did not specify the maximum number of threads to use, the generated MEX function executes the loop iterations in parallel on the maximum number of available cores.

Accelerate MATLAB Algorithms That Use Parallel for-loops (parfor) Specifying the Maximum Number of Threads

This example shows how to specify the maximum number of threads to use for a parfor-loop. Because you specify the maximum number of threads to use, the generated MEX function executes the loop iterations in parallel on as many cores as available, up to the maximum number that you specify. If you specify more threads than there are cores available, the MEX function uses the available cores.

- 1 Write a MATLAB function, `specify_num_threads`, that uses one input to specify the maximum number of threads to execute a parfor-loop in the generated MEX function. For example:

```
function y = specify_num_threads(u) %#codegen
    y = ones(1,100);
    % u specifies maximum number of threads
    parfor (i = 1:100,u)
        y(i) = i;
    end
end
```

- 2 Generate a MEX function for `specify_num_threads`. Use `-args {0}` to specify that input `u` is a scalar double. Use `-report` to generate a code generation report. At the MATLAB command line, enter:

```
codegen -report specify_num_threads -args {0}
```

`codegen` generates a MEX function, `specify_num_threads_mex`, in the current folder.

- 3 Run the MEX function, specifying that it try to run in parallel on four threads. At the MATLAB command line, enter:

```
specify_num_threads_mex(4)
```

The generated MEX function runs on up to four cores. If less than four cores are available, the MEX function runs on the maximum number of cores available at the time of the call.

Troubleshooting parfor-loops

What Causes Errors About the Use of Global Structures in Parallel Regions?

- The body of the parfor-loop contains global or persistent variable declarations. parfor does not support such declarations.
- The parfor-loop contains a call to rand.
- Local variables use more memory than the specified stack size. When this occurs, MATLAB Coder moves the local variables to a static area and accesses them using a pointer in a global structure. MATLAB Coder does not support the use of global structures in parallel regions. If possible, increase the stack size.

If you are using...	Action	For More Information
A MATLAB Coder project	In the Project Settings dialog box, on the Advanced tab, set Inline stack limit to the new limit.	“Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-28
codegen at the command line with a configuration object	Create a coder.CodeConfig or coder.EmbeddedCodeConfig object, as applicable, and set the InlineStackLimit parameter to the new limit.	“Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-29

Compiler Does Not Support OpenMP

The MATLAB Coder software uses the Open Multi-Processing (OpenMP) application interface to support shared-memory, multicore code generation. This allows you to use parfor to generate MEX functions that run in parallel on multiple cores on a desktop. For MEX functions, OpenMP is enabled by default. If your compiler does not support OpenMP, MATLAB Coder generates a warning.

Install a compiler that supports OpenMP. You can use supported compilers except Microsoft Visual Studio SDK, Open Watcom, LCC,

and Apple Xcode with Clang. For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Accelerating Simulation of Bouncing Balls

This example shows how to accelerate MATLAB algorithm execution using a generated MEX function. It uses the 'codegen' command to generate a MEX function for a complicated application that uses multiple MATLAB files. You can use 'codegen' to check that your MATLAB code is suitable for code generation and, in many cases, to accelerate your MATLAB algorithm. You can run the MEX function to check for run-time errors.

Prerequisites

To run this example, you must install a C compiler and set it up using the 'mex-setup' command. For more information, see [Setting Up Your C Compiler](#).

Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will contain only the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_bouncing_balls');
```

About the 'run_balls' Function

The run_balls.m function takes a single input to specify the number of bouncing balls to simulate. The simulation runs and plots the balls bouncing until there is no energy left and returns the state (positions) of all the balls.

```
type run_balls
```

```
% balls = run_balls(n)
% Given 'n' number of balls, run a simulation until the balls come to a
% complete halt (or when the system has no more kinetic energy).
function balls = run_balls(n) %#codegen

% Copyright 2010-2011 The MathWorks, Inc.
```



```
% Seeding the random number generator will guarantee that we get
% precisely the same simulation every time we call this function.
old_settings = rng(1283,'V4');

% The 'cdata' variable is a matrix representing the colordata bitmap which
% will be rendered at every time step.
cdata = zeros(400,600,'uint8');

% Setup figure windows
im = setup_figure_window(cdata);

% Get the initial configuration for 'n' balls.
balls = initialize_balls(cdata, n);

energy = 2; % Something greater than 1
while energy > 1
    % Clear the bitmap
    cdata(:, :) = 0;
    % Apply one iteration of movement
    [cdata,balls,energy] = step_function(cdata,balls);
    % Render the current state
    cdata = draw_balls(cdata, balls);
    refresh_image(im, cdata);
end

% Restore RNG settings.
rng(old_settings);
```

Generate the MEX Function

First, generate a MEX function using the command `codegen` followed by the name of the MATLAB file to compile. Pass an example input (`-args 0`) to indicate that the generated MEX function will be called with an input of type double.

```
codegen run_balls -args 0
```

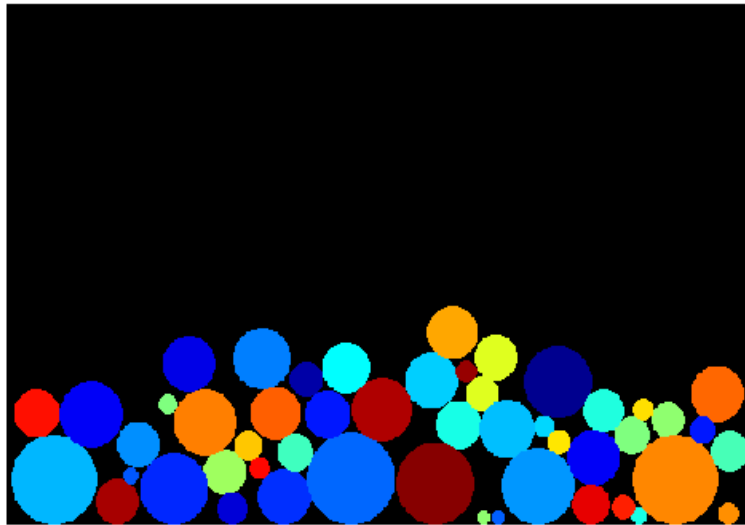
The `'run_balls'` function calls other MATLAB functions, but you need to specify only the entry-point function when calling `'codegen'`.

By default, 'codegen' generates a MEX function named 'run_balls_mex' in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Compare Results

Run and time the original 'run_balls' function followed by the generated MEX function.

```
tic, run_balls(50); t1 = toc;  
tic, run_balls_mex(50); t2 = toc;
```



Estimated speed up is:

```
fprintf(1, 'Speed up: x ~%2.1f\n', t1/t2);
```

```
Speed up: x ~6.1
```

Clean Up

Remove files and return to original folder

Run Command: Cleanup

```
cleanup
```


Calling C/C++ Functions from Generated Code

- “MATLAB® Coder™ Interface to C/C++ Code” on page 23-2
- “Call External C/C++ Functions” on page 23-7
- “Return Multiple Values from C Functions” on page 23-9
- “How MATLAB® Coder™ Infers C/C++ Data Types” on page 23-10

MATLAB Coder Interface to C/C++ Code

In this section...
“How to Call C/C++ Code from Generated Code” on page 23-2
“Why Call C/C++ Functions from Generated Code?” on page 23-2
“Call External C/C++ Functions” on page 23-3
“Pass Arguments by Reference to External C/C++ Functions” on page 23-3
“Manipulate C Data” on page 23-5

How to Call C/C++ Code from Generated Code

MATLAB Coder provides a set of functions for:

- Calling external C/C++ code from generated code (see “Call External C/C++ Functions” on page 23-3)
- Passing arguments by reference to C/C++ code (see “Pass Arguments by Reference to External C/C++ Functions” on page 23-3)
- Manipulating C/C++ data (see “Manipulate C Data” on page 23-5)

By using these functions, you gain unrestricted access to external C/C++ code. Misuse of these functions or errors in your C/C++ code can destabilize MATLAB when generating MEX functions.

Why Call C/C++ Functions from Generated Code?

Call C/C++ functions from generated code when you want to:

- Use legacy C/C++ code
- Use your own optimized C/C++ functions instead of generated code.
- Interface your libraries and hardware with MATLAB functions.

Call External C/C++ Functions

Use the `coder.ceval` function to call external C/C++ functions. `coder.ceval` passes function input and output arguments to C/C++ functions either by value or by reference.

You must define these called functions in external C/C++ source files or in C/C++ libraries. You then need to include C/C++ source files, libraries, object files, and header files in the compilation to configure your environment.

Pass Arguments by Reference to External C/C++ Functions

By default, `coder.ceval` passes arguments by value to the C/C++ function whenever C/C++ supports passing arguments by value. The following constructs allow you to pass MATLAB variables as arguments by reference to external C/C++ functions:

- `coder.ref` — pass value by reference
- `coder.rref` — pass read-only value by reference
- `coder.wref` — pass write-only value by reference

These constructs offer the following benefits:

- Passing values by reference optimizes memory use.

When you pass arguments by value, MATLAB Coder passes a copy of the value of each argument to the C/C++ function to preserve the original values. When you pass arguments by reference, MATLAB Coder does not copy values. The memory savings can be significant if you need to pass large matrices to the C/C++ function.

- Passing write-only values by reference allows you to return multiple outputs.

Use `coder.wref` to return multiple outputs from your C/C++ function, including arrays and matrices. Otherwise, the C/C++ function can return only a single scalar value through its return statement.

Do not store pointers that you pass to C/C++ functions because MATLAB Coder optimizes the code based on the assumption that you do not store the

addresses of these variables. Storing the addresses might invalidate our optimizations leading to incorrect behavior. For example, if a MATLAB function passes a pointer to an array using `coder.ref`, `coder.rref`, or `coder.wref`, then the C/C++ function can modify the data in the array—but you should not store the pointer for future use.

When you pass arguments by reference using `coder.rref`, `coder.wref`, and `coder.ref`, the corresponding C/C++ function signature must declare these variables as pointers of the same data type. Otherwise, the C/C++ compiler generates a type mismatch error.

For example, suppose your MATLAB function calls an external C function `ctest`:

```
function y = fcn()
u = pi;

y = 0;
y = coder.ceval('ctest',u);
```

Now suppose the C function signature is:

```
real32_T ctest(real_T *a)
```

When you compile the code, you get a type mismatch error because `coder.ceval` calls `ctest` with an argument of type `double` when `ctest` expects a pointer to a double-precision, floating-point value.

Match the types of arguments in `coder.ceval` with their counterparts in the C function. For instance, you can fix the error in the previous example by passing the argument by reference:

```
y = coder.ceval('ctest', coder.rref(u));
```

You can pass a reference to an element of a matrix. For example, to pass the second element of the matrix `v`, you can use the following code:

```
y = coder.ceval('ctest', coder.ref(v(1,2)));
```


Manipulate C Data

The construct `coder.opaque` allows you to manipulate C/C++ data that a MATLAB function does not recognize. You can store the opaque data in a variable or structure field and pass it to, or return it from, a C/C++ function using `coder.ceval`.

Declaring Opaque Data

The following example uses `coder.opaque` to declare a variable `f` as a `FILE *` type.

```
% This example returns its own source code by using
% fopen/fread/fclose.
function buffer = filetest
%#codegen

% Declare 'f' as an opaque type 'FILE *'
f = coder.opaque('FILE *', 'NULL');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));

% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
    % By default, MATLAB converts constant values
    % to doubles in generated code
    % so explicit type conversion to in32 is inserted.
    n = coder.ceval('fread', coder.ref(buffer(i)), int32(1), ...
        int32(numel(buffer)), f);
    i = i + n;
end
coder.ceval('fclose',f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB string
function y = cstring(x)
```

```
y = [x char(0)];

% Remove character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
j = 1;
for i = 1:numel(buffer)
    if buffer(i) ~= char(13)
        buffer(j) = buffer(i);
        j = j + 1;
    end
end
buffer(i) = 0;
```

Call External C/C++ Functions

In this section...

“Workflow for Calling External C/C++ Functions” on page 23-7

“Best Practices for Calling C/C++ Code from Generated Code” on page 23-8

Workflow for Calling External C/C++ Functions

To call external C/C++ functions from generated code:

- 1 Write your C/C++ functions in external source files or libraries.
- 2 Create header files, if required.

The header file defines the data types used by the C/C++ functions that MATLAB Coder generates in code, as described in “Mapping MATLAB Types to C/C++” on page 23-10.

Tip One way to add these type definitions is to include the header file `tmwtypes.h`, which defines general data types supported by MATLAB. This header file is in `matlabroot/extern/include`. Check the definitions in `tmwtypes.h` to determine if they are compatible with your target. If not, define these types in your own header files.

- 3 In your MATLAB function, add calls to `coder.ceval` to invoke your external C/C++ functions.

You need one `coder.ceval` statement for each call to a C/C++ function. In your `coder.ceval` statements, use `coder.ref`, `coder.rref`, and `coder.wref` constructs as required (see “Pass Arguments by Reference to External C/C++ Functions” on page 23-3).

- 4 Include the custom C/C++ functions in the build. See “Custom C/C++ Code Integration” on page 21-12.
- 5 Check for compilation warnings about data type mismatches.

Perform this check so that you catch type mismatches between C/C++ and MATLAB (see “How MATLAB® Coder™ Infers C/C++ Data Types” on page 23-10).

6 Generate code and fix errors.

7 Run your application.

Best Practices for Calling C/C++ Code from Generated Code

The following are recommended practices when calling C/C++ code from generated code.

- **Start small.** — Create a test function and learn how `coder.ceval` and its related constructs work.
- **Use separate files.** — Create a file for each C/C++ function that you call. Make sure that you call the C/C++ functions with suitable types.
- In a header file, declare a function prototype for each function that you call, and include this header file in the generated code. For more information, see “Custom C/C++ Code Integration” on page 21-12.

Return Multiple Values from C Functions

The C language restricts functions from returning multiple outputs; instead, they return only a single, scalar value. The constructs `coder.ref` and `coder.wref` allow MATLAB functions to exchange multiple outputs with the external C functions that they call.

For example, suppose you write a MATLAB function `foo` that takes two inputs `x` and `y` and returns three outputs `a`, `b`, and `c`. In MATLAB, you call this function as follows:

```
[a, b, c] = foo (x, y)
```

If you rewrite `foo` as a C function, you cannot return `a`, `b`, and `c` through the `return` statement. You can create a C function with multiple pointer type input arguments, and pass the output parameters by reference. For example:

```
foo(real_T x, real_T y, real_T *a, real_T *b, real_T *c)
```

Then you can call the C function with multiple outputs from a MATLAB function using `coder.wref` constructs:

```
coder.ceval ('foo', x, y, ...  
            coder.wref(a), coder.wref(b), coder.wref(c));
```

Similarly, suppose that one of the outputs `a` is also an input argument. In this case, create a C function with multiple pointer type input arguments, and pass the output parameters by reference. For example:

```
foo(real_T *a, real_T *b, real_T *c)
```

Then call the C function from a MATLAB function using `coder.wref` and `coder.rref` constructs:

```
coder.ceval ('foo', coder.rref(a), coder.wref(b), coder.wref(c));
```

How MATLAB Coder Infers C/C++ Data Types

In this section...

- “Mapping MATLAB Types to C/C++” on page 23-10
- “Mapping embedded.numerictypes to C/C++” on page 23-11
- “Mapping Arrays to C/C++” on page 23-12
- “Mapping Complex Values to C/C++” on page 23-12
- “Mapping Structures to C/C++ Structures” on page 23-13
- “Mapping Strings to C/C++” on page 23-14
- “Mapping Multiword Types to C/C++” on page 23-14

Mapping MATLAB Types to C/C++

The C/C++ type associated with a MATLAB variable or expression is based on the following properties:

- Class
- Size
- Complexity

The following translation table shows the MATLAB types supported for code generation, and how MATLAB Coder infers the generated code types.

MATLAB Type	C/C++ Type	C/C++ Reference Type
int8	int8_T	int8_T *
int16	int16_T	int16_T *
int32	int32_T	int32_T *
uint8	uint8_T	uint8_T *
uint16	uint16_T	uint16_T *
uint32	uint32_T	uint32_T *
double	real_T	real_T *

MATLAB Type	C/C++ Type	C/C++ Reference Type
single	real32_T	real32_T *
char	char	char *
logical	boolean_T	boolean_T *
fi	numericaltype also influences the C/C++ type. Integer type varies according to the MATLAB fixed-point type, as described in “Mapping embedded.numerictypes to C/C++” on page 23-11.	
struct	The MATLAB Coder software translates structures to C/C++ types field-by-field. See “Mapping Structures to C/C++ Structures” on page 23-13 .	
complex	See “Mapping embedded.numerictypes to C/C++” on page 23-11.	
Function handles	Not supported.	
Multiword types	See “Mapping Multiword Types to C/C++” on page 23-14.	

Mapping embedded.numerictypes to C/C++

The following translation table shows how MATLAB Coder infers integer types from fixed-point objects. In the first column, the fixed-point types are specified by the Fixed-Point Designer function `numerictype`:

`numerictype(signedness, word length, fraction length)`

The MATLAB for code generation integer type is the next larger target word size that can store the fixed-point value, based on its word length. The sign of the integer type matches the sign of the fixed-point type.

embedded.numerictype	C/C++ Type	C/C++ Reference Type
<code>numerictype(1, 16, 15)</code>	int16_T	int16_T *
<code>numerictype(1, 13, 10)</code>	int16_T	int16_T *

embedded.numericType	C/C++ Type	C/C++ Reference Type
numericType(0, 19, 15)	uint32_T	uint32_T *
numericType(1, 8, 7)	int8_T	int8_T *

Mapping Arrays to C/C++

The following translation table shows how MATLAB Coder determines array types and sizes in generated code. In the first column, the arrays are specified by the MATLAB function `zeros`:

`zeros(number of rows, number of columns, data type)`

MATLAB array data is laid out in column major order.

Array	C/C++ Type	C/C++ Reference Type
<code>zeros(10, 5, 'int8')</code>	int8_T *	int8_T *
<code>zeros(5, 10, 'int8')</code>	int8_T *	int8_T *
<code>zeros(3, 7)</code>	real_T *	real_T *
<code>zeros(10, 1, 'single')</code>	real32_T *	real32_T *

Mapping Complex Values to C/C++

The following translation table shows how the MATLAB Coder infers complex values in generated code.

Complex	C/C++ Type	C/C++ Reference Type
complex int8	cint8_T	cint8_T *
complex int16	cint16_T	cint16_T *
complex int32	cint32_T	cint32_T *
complex uint8	cuint8_T	cuint8_T *

Complex	C/C++ Type	C/C++ Reference Type
complex uint16	cuint16_T	cuint16_T *
complex uint32	cuint32_T	cuint32_T *
complex double	creal_T	creal_T *
complex single	creal32_T	creal32_T *

The MATLAB Coder software defines each complex value as a structure with a real component `re` and an imaginary component `im`, as in this example from `tmwtypes.h`:

```
typedef struct {
    real32_T re; /* Real component*/
    real32_T im; /* Imaginary component*/
} creal32_T;
```

MATLAB Coder uses the names `re` and `im` in generated code to represent the components of complex numbers. For example, suppose you define a variable `x` of type `creal32_T`. The generated code references the real component as `x.re` and the imaginary component as `x.im`.

If your C/C++ library requires a different representation, you can define your own versions of MATLAB Coder complex types, but you *must* use the names `re` for the real components and `im` for the imaginary components in your definitions.

The MATLAB Coder software represents a matrix of complex numbers as an array of structures.

Mapping Structures to C/C++ Structures

The MATLAB Coder software translates structures to C/C++ types field-by-field. The order of the field items is preserved as given in MATLAB. To control the name of the generated C/C++ structure type, or provide a definition, use the `coder.cstructname` function.

Note If you are not using dynamic memory allocation, arrays in structures translate into single-dimension arrays, not pointers.

Mapping Strings to C/C++

The MATLAB Coder software translates MATLAB strings to C/C++ character matrices. Character matrices cannot be used as substitutes for C/C++ strings because they are not null terminated. You can terminate a MATLAB string with a null character by appending a zero to the end of the string: ['sample string' 0]. A single character translates to a C/C++ char type, not a C/C++ string.

Caution Failing to null-terminate your MATLAB strings might cause C/C++ code to crash without compiler errors or warnings.

Mapping Multiword Types to C/C++

The MATLAB Coder software translates multiword types to structure types that contain an array of integers. The array dimensions depend on the long type on the target hardware. For example, for a 128-bit fixed-point type, if the long type on the target hardware is 32-bits, MATLAB Coder generates a structure with an array of four 32-bit integers.

```
typedef struct
{
    uint32_T chunks[4];
} uint128m_T;
```

If the long type on the target hardware is 64-bits, MATLAB Coder generates a structure with an array of two 64-bit integers.

```
typedef struct
{
    uint64_T chunks[2];
} uint128m_T;
```

A

- arguments
 - limit on number for code generation from MATLAB 13-19

C

- C/C++ code generation for supported functions 4-1
- code files
 - packaging 19-196
 - porting 19-196
- code generation from MATLAB
 - benefits of 2-2
 - best practices for working with variables 5-3
 - calling local functions 13-9
 - calling MATLAB functions 13-11
 - calling MATLAB functions using feval 13-16
 - characters 6-6
 - communications system toolbox System objects 3-7
 - compilation directive `%#codegen` 13-8
 - computer vision system toolbox System objects 3-2
 - converting mxArray to known types 13-18
 - declaring MATLAB functions as extrinsic functions 13-12
 - defining persistent variables 5-10
 - defining variables 5-2
 - defining variables by assignment 5-3
 - dsp system toolbox System objects 3-13
 - eliminating redundant copies of uninitialized variables 5-7
 - how it resolves function calls 13-2
 - initializing persistent variables 5-10
 - limit on number of function arguments 13-19
 - pragma 13-8
 - resolving extrinsic function calls during simulation 13-16
 - resolving extrinsic function calls in generated code 13-17
 - rules for defining uninitialized variables 5-7
 - setting properties of indexed variables 5-6
 - supported toolbox functions 13-10
 - using type cast operators in variable definitions 5-6
 - variables, complex 6-4
 - when not to use 2-2
 - when to use 2-2
 - which features to use 2-4
 - working with mxArray 13-17
- code generation readiness 17-4
- code generation report keyboard shortcuts
 - codegen 19-193
- codegen
 - code generation report keyboard shortcuts 19-193
 - generating code for more than one entry-point file 19-75
 - global data 19-81
- `coder.extrinsic` 13-12
- `coder.nullcopy`
 - uninitialized variables 5-7
- comments in generated code
 - codegen 19-184
 - MATLAB Coder 19-88
- communications system toolbox System objects supported for code generation from MATLAB 3-7
- computer vision system toolbox System objects supported for code generation from MATLAB 3-2
- configuration objects
 - codegen 19-29
- controlling run-time checks
 - MATLAB Coder 22-18
- cross-development
 - packaging files for 19-196

D

- debugging run-time errors
 - MATLAB 18-9
- defining uninitialized variables
 - rules 5-7
- defining variables
 - for C/C++ code generation 5-3
- design considerations
 - when writing MATLAB Code for code generation 2-6 17-28
- dsp system toolbox System objects
 - supported for code generation from MATLAB 3-13

E

- eliminating redundant copies of function inputs 19-66 22-15
- extrinsic functions 13-12

F

- functions
 - limit on number of arguments for code generation 13-19
- Functions supported for C/C++ code generation 4-1
 - alphabetical list 4-2
 - arithmetic operator functions 4-81
 - bit-wise operation functions 4-81
 - casting functions 4-82
 - Communications System Toolbox functions 4-82
 - complex number functions 4-82
 - Computer Vision System Toolbox functions 4-83
 - data and file management functions 4-84
 - data type functions 4-85
 - derivative and integral functions 4-85
 - discrete math functions 4-85

- error handling functions 4-86
- exponential functions 4-86
- filtering and convolution functions 4-87
- Fixed-Point Designer functions 4-87
- histogram functions 4-96
- Image Processing Toolbox functions 4-96
- input and output functions 4-98
- interpolation and computational geometry functions 4-99
- linear algebra functions 4-99
- logical operator functions 4-99
- MATLAB Compiler functions 4-100
- MATLAB Desktop environment functions 4-100
- matrix/array functions 4-100
- nonlinear numerical methods 4-104
- polynomial functions 4-105
- relational operator functions 4-105
- rounding and remainder functions 4-105
- set functions 4-106
- signal processing functions 4-106
- Signal Processing Toolbox functions 4-107
- special value functions 4-111
- specialized math functions 4-112
- statistical functions 4-112
- string functions 4-113
- structure functions 4-114
- trigonometric functions 4-114

- Functions supported for MEX and C/C++ code generation
 - categorized list 4-79

G

- generating code for more than one entry-point file codegen 19-75
- generating traceable code
 - MATLAB Coder 19-88
- global data
 - codegen 19-81

H

how to disable run-time checks
MATLAB Coder 22-19

I

indexed variables
 setting properties for code generation from
 MATLAB 5-6
initialization
 persistent variables 5-10

M

MATLAB
 debugging run-time errors 18-9
 features not supported for code
 generation 2-13
MATLAB code analyzer
 using with MATLAB for code generation 17-4
MATLAB Coder
 combining property specifications 19-58
 controlling run-time checks 22-18
 eliminating redundant copies of function
 inputs 19-66 22-15
 how to disable run-time checks 22-19
 inlining functions 19-65 22-14
 specifying build configuration
 parameters 19-28
 specifying general properties of primary
 inputs 19-58
 when to disable run-time checks 22-19
MATLAB for code generation
 using the MATLAB code analyzer 17-4
 variable types 5-18
MATLAB functions
 and generating code for mxArray 13-17
mxArrays
 converting to known types 13-18
 for code generation from MATLAB 13-17

O

optimizing generated code
 unrolling for-loops 19-63 22-12

P

parfor-loops
 break 22-25
 broadcast variables 22-32
 classification of variables 22-29
 global variables 22-26
 nested loops 22-25
 persistent variables 22-26
 reduction assignments 22-33
 reduction assignments, associativity 22-37
 reduction assignments, commutativity 22-37
 reduction variables 22-32
 return 22-25
 sliced variables 22-30
 temporary variables 22-38
persistent variables
 defining for code generation from
 MATLAB 5-10
 initializing for code generation from
 MATLAB 5-10

R

readability
 codegen 19-184
 MATLAB Coder 19-88

S

signal processing functions
 for C/C++ code generation 4-107
specifying build configuration parameters
 codegen 19-29
 MATLAB Coder 19-28

T

- traceability
 - codegen 19-184
 - MATLAB Coder 19-88
- type cast operators
 - using in variable definitions 5-6

U

- uninitialized variables
 - eliminating redundant copies in generated code 5-7

V

- validating code
 - codegen 19-184

MATLAB Coder 19-88

- variable types supported for code generation
 - from MATLAB 5-18

- variables
 - eliminating redundant copies in C/C++ code generated from MATLAB 5-7

Variables

- defining by assignment for code generation
 - from MATLAB 5-3
- defining for code generation from MATLAB 5-2

W

- when to disable run-time checks
 - MATLAB Coder 22-19