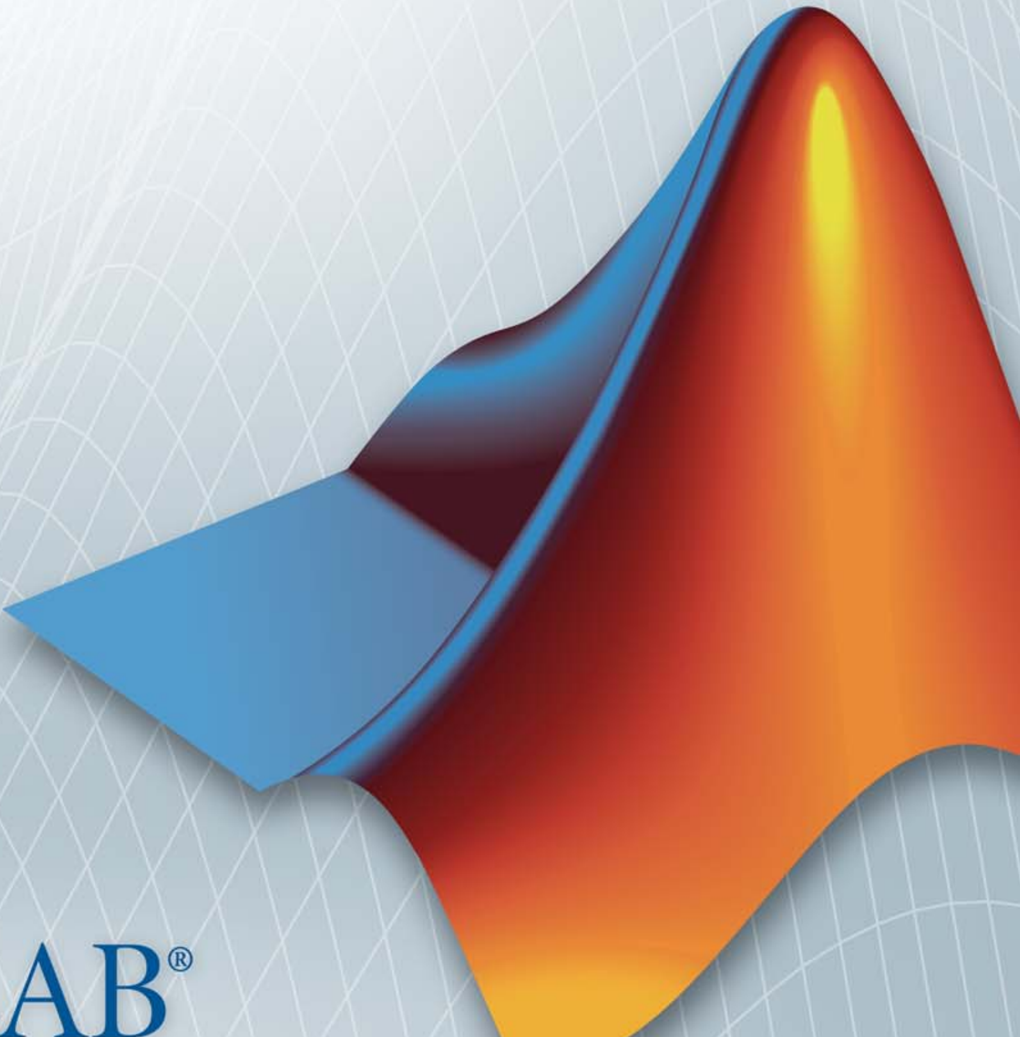
MATLAB® Coder™

User's Guide

R2011b



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB® Coder™ User's Guide

Revision History

April 2011	Online only	New for Version 2 (R2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)

Contents

1 About MATLAB Coder

2 Bug Reports

3 Setting Up a MATLAB® Coder Project

4 Preparing MATLAB Code for C/C++ Code Generation

5 Testing MEX Functions in MATLAB

6 Generating C/C++ Code from MATLAB Code

About MATLAB Coder

Product Overview

In this section...

About MATLAB Coder

MATLAB® Coder™ generates standalone C and C++ 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.

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.

See Also

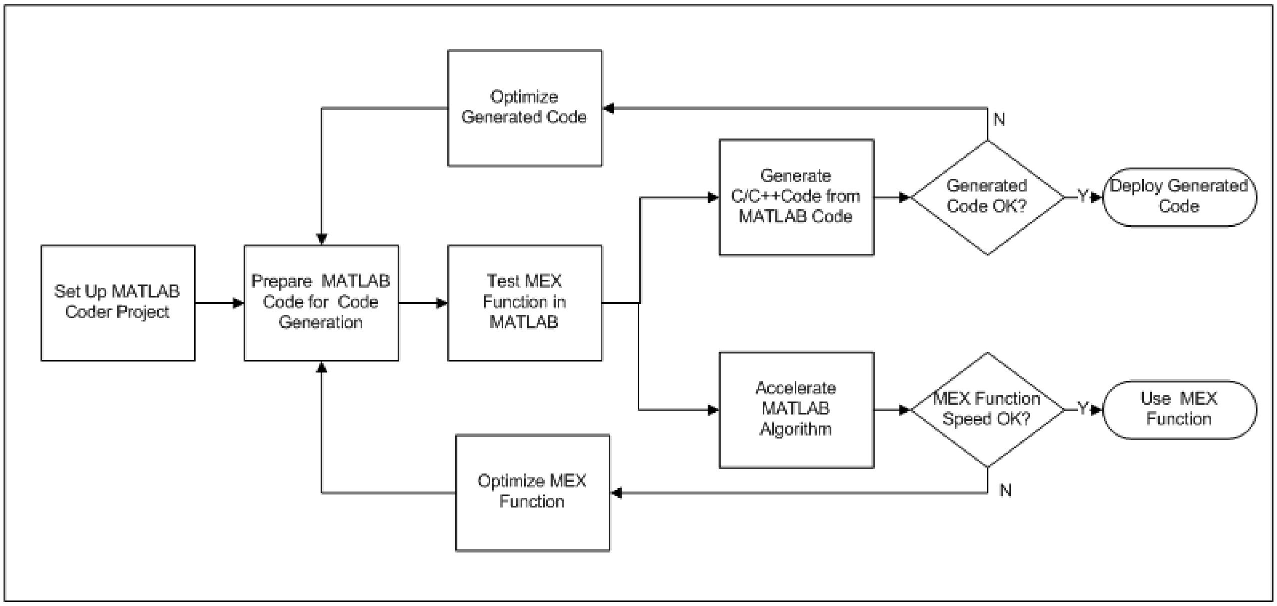
- "MATLAB Tutorials" in the Embedded Coder documentation.

Code Generation for Fixed-Point Algorithms

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

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

Code Generation Workflow


Optimize Generated Code
N
Generate C/C++Code from MATLAB Code
Generated Code OK?
Y
Deploy Generated Code
Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Test MEX Function in MATLAB
Accelerate MATLAB Algorithm
MEX Function Speed OK?
Y
Use MEX Function
Optimize MEX Function
N

See Also

- Chapter 3, “Setting Up a MATLAB® Coder Project”
- Chapter 4, “Preparing MATLAB Code for C/C++ Code Generation”
- Chapter 5, “Testing MEX Functions in MATLAB”
- Chapter 6, “Generating C/C++ Code from MATLAB Code”
- Chapter 7, “Accelerating MATLAB Algorithms”

Bug Reports

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

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

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

Setting Up a MATLAB Coder Project

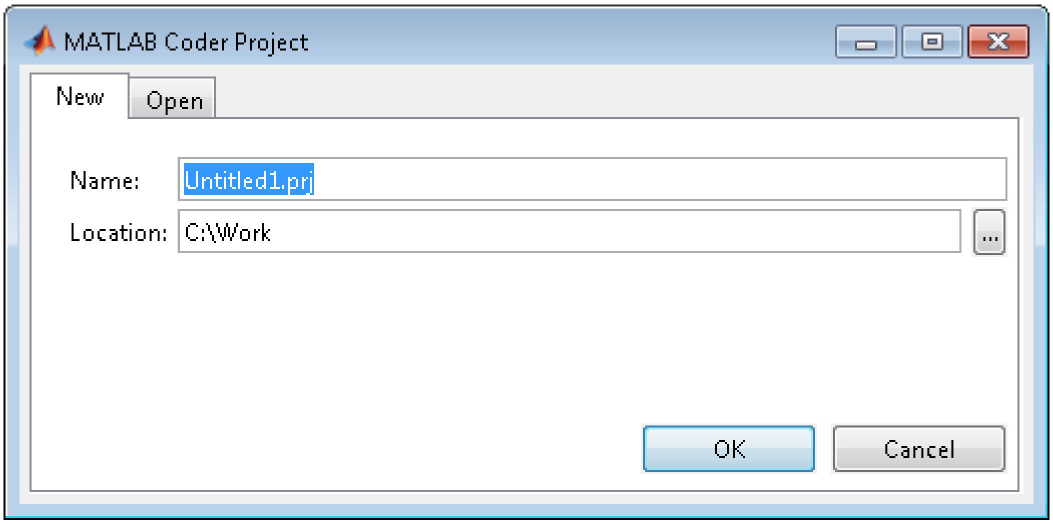
MATLAB Coder Project Set Up Workflow

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), C/C++ library, or C/C++ executable.

Creating a New Project

1 At the MATLAB command line, enter:

```
coder
```



MATLAB Coder Project

Name: Untitled1.prj

Location: C:\Work

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 MathWorks® recommends that the path 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: .

Alternatively, if you already have a MATLAB Coder open, in the upper-right corner of the project, click the **Actions** icon ( ) and select `New Project`.

Opening an Existing Project

1 At the MATLAB command line, enter `coder`.

2 In the MATLAB Coder 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**.

Alternatively, 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. MathWorks recommends that the path 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.

.

Note If you are using preconditions (see "Defining Input Properties Programmatically in the MATLAB File" on page 6-42) to define input types, project files do not currently recognize them. Consider using `codegen` instead, or reentering types in the project.

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 3-7.

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 subfunctions 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 6-30

How to Specify an Input Definition in a Project

Specify an input definition in your MATLAB Coder project using one of the following methods:

- Define by Example
- Define Type
- Define Constant

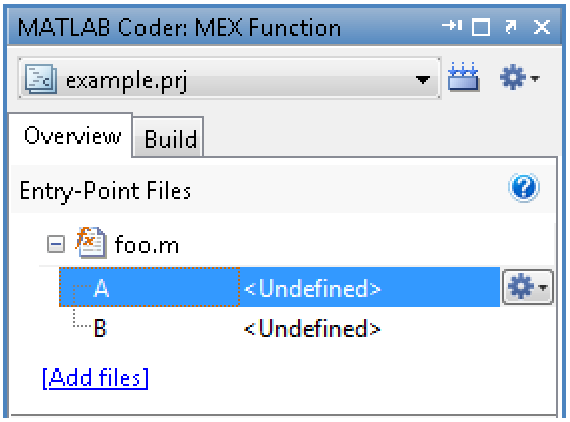
Alternatively, specify input definitions at the command line and then use the `codegen` function to generate code. For more information, see “Primary Function Input Specification” on page 6-30.

Defining Input Parameters by Example in a Project

In this section...

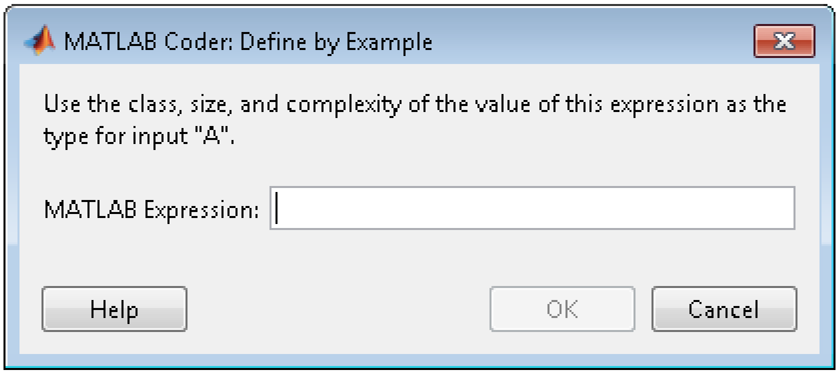
How to Define an Input Parameter by Example

1 On the MATLAB Coder project **Overview** tab, select the input parameter that you want to define.



2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From this menu, select `Define by Example`.



Use the class, size, and complexity of the value of this expression as the type for input "A".

MATLAB Expression:

4 In the Define by Example dialog box, 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

To specify a `1-by-4` vector of unsigned 16-bit integers, in the Define by Example dialog box, enter:

```
zeros(1,4,'uint16')
```

To specify that an input is a double-precision scalar, simply enter `0`.

Optionally, after specifying the input type, you can specify that the input is variable size.

1 To specify the input type, in the Define by Example dialog box, enter:

```
zeros(1,4,'uint16')
```

2 To specify that the input is variable size.

   a On the project **Overview** tab, select the input.

   b To the right of `u`, click the **Actions** icon to open the context menu.

   c From the menu, select `Edit Type`.

   d In the Define Type dialog box, click the **Actions** icon to open the context menu.

e From the menu, select `Make Sizes Variable`.

The size of variable `u` changes from `1x4` to `1x:4`. The colon, `:` indicates that the second dimension is variable size with an upper bound of `4`.

Alternatively, you can specify that the input is variable size by using the `coder.newtype` function. In the Define by Example dialog box, enter the following MATLAB expression:

```
coder.newtype('uint16',[1 4],[0 1])
```

Specifying an Enumerated Type Input Parameter by Example

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 Define by Example dialog box, 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 Toolbox software.

To specify a signed fixed-point type with a word length of 8 bits, and a fraction length of 3 bits, in the Define by Example dialog box, 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 `fimath`, MATLAB Coder uses the global `fimath`. For more information, see “Working with the Global fimath” in the Fixed-Point Toolbox documentation.

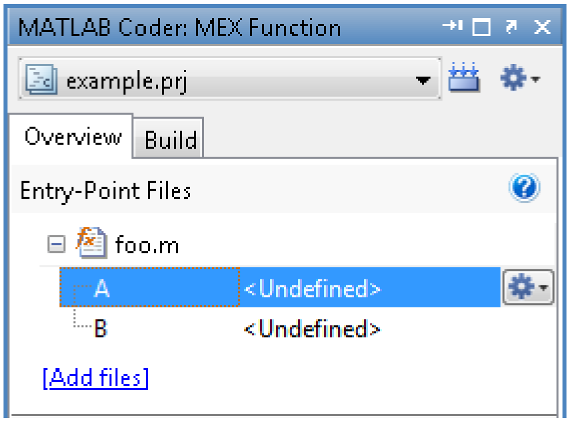
Optionally, modify the fixed-point data type definition. See “Editing a Fixed-Point Input Parameter Type” on page 3-17.

Defining or Editing Input Parameter Type in a Project

In this section...
"How to Define or Edit an Input Parameter Type" on page 3-12
"Editing a Fixed-Point Input Parameter Type" on page 3-17
"Specifying Structures" on page 3-21

How to Define or Edit an Input Parameter Type

1 On the MATLAB Coder project **Overview** tab, select the input parameter that you want to define.



2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From the menu, select `Define Type` or `Edit Type`, as applicable.

4 In the Define Type dialog box, from the **Class** drop-down list, select a class. The default class is `double`.

Note To specify an enumerated type, enter its name in the **Class** field. Define the enumerated type in a file on the MATLAB path. For more information, see "Code Generation for Enumerated Types" on page 6-98.

---

5. In the **Size** field, enter the size.

   Note If you enter an invalid size, MATLAB Coder highlights the **Size** field in red to indicate that you must correct the size specification.

   ---

6. Optionally, to edit the type definition further, to the right of the **Size** field, click the **Actions** icon.

   MATLAB Coder displays the following menu options:

   - Edit Size Vector Definition

     Select this option to define the size of each dimension of the input. For more information, see "Edit Size Vector Definition" on page 3-14.

   - Make Sizes Variable

     For nonscalar inputs, select this option to make all the dimensions variable size.

     Note For any dimension with a size of `0` or `1`, selecting **Make Sizes Variable** has no effect. However, you can manually make a dimension with a size of `1` variable-size by selecting **Edit Size Vector Definition** and entering a size of `:1` for this dimension.

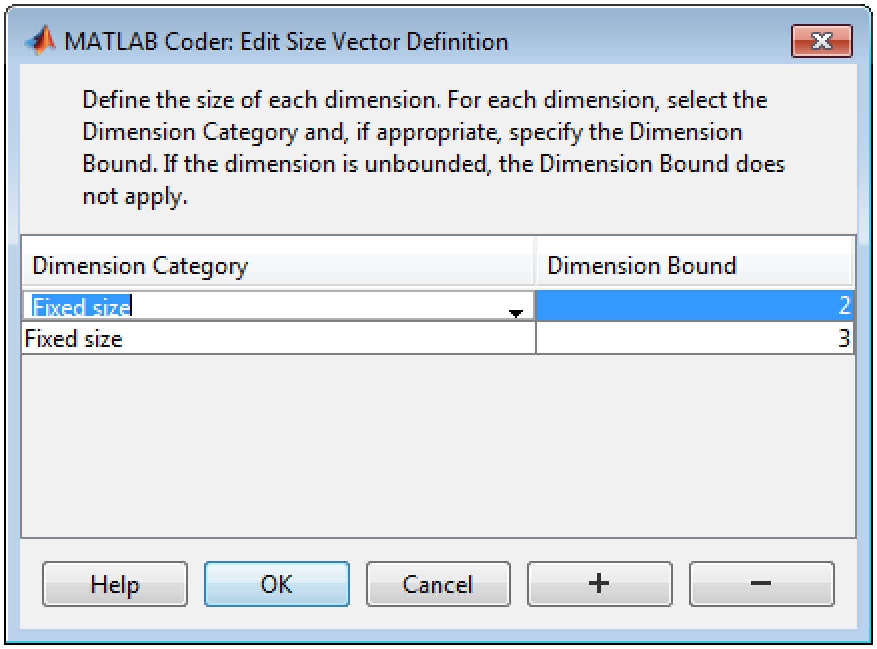
   - Make Sizes Fixed

     If some or all the dimensions of an input are variable size, select this option to make all the dimensions, except those that are unbounded, fixed in size.

7. Optionally, for numeric types, select **Complex** to make the parameter a complex type. By default, inputs are real.

Edit Size Vector Definition

The Edit Size Vector Definition dialog box provides information about each dimension of the input. For example, here the input is a fixed-size 2x3 matrix.


MATLAB Coder: Edit Size Vector Definition

Define the size of each dimension. For each dimension, select the Dimension Category and, if appropriate, specify the Dimension Bound. If the dimension is unbounded, the Dimension Bound does not apply.

Dimension Category	Dimension Bound
Fixed size	2
Fixed size	3

You can modify the size vector definition as described in the following table.

To...	Do this...
Specify a variable-size dimension with fixed upper bounds	**1** Select the dimension that you want to change. **2** Set the **Dimension Category** to `Variable with specified upper bounds`. **Note** If the size of the dimension is `0` or `1`, it is always fixed in size, so selecting `Variable with specified upper bounds` has no effect. **3** In the **Dimension Bound** field, enter the upper bound value or use the up and down arrows to adjust the value.
Specify a variable-size dimension with no upper bounds	**1** Select the dimension that you want to change. **2** Set the **Dimension Category** to `Unbounded size`. **Note** This action disables the **Dimension Bound** field for the selected dimension.
Add a dimension	**1** Select the dimension below which you want to add a new dimension. **2** Click **+**. By default, the new dimension is fixed size with an upper bound of `1`.
Remove a dimension	Select the dimension to remove and click **–**.

Specifying Fixed-Size Input Parameters by Type

To specify a `1-by-4` vector of unsigned 16-bit integers, in the Define Type dialog box, set:

1 **Class** to `uint16`.

2 **Size** to `1x4`.

Specifying Variable-Size Input Parameters by Type

To specify that an input is a variable-size `1-by-4` vector of unsigned 16-bit integers:

1 In the Define Type dialog box, set:

   a **Class** to `uint16`.

   b **Size** to `1x4`.

2 To the right of the Size field, click the **Actions** icon to open the context menu.

3 From the menu, select `Make Sizes Variable`.

   The size of variable `u` changes from `1x4` to `1x:4`. The `:` indicates that the second dimension is variable size with an upper bound of `4`.

   Note Alternatively, in the **Size** field, enter `1x:4` to specify a variable-size input with an upper bound of `4`. If the size of the dimension is `0` or `1`, it is always fixed in size.

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 Define Type dialog box **Class** type field, enter `MyColors`.

Specifying a Fixed-Point Input Parameter by Type

To specify fixed-point inputs, you must install Fixed-Point Toolbox software.

1 In the Define Type dialog box, set **Class** to `embedded.fi`.

   MATLAB Coder sets the type of input `u` to `embedded.fi(1x1)`. By default, if there is no local `fimath`, uses the global `fimath`. For more information, see "Working with the Global fimath" in the Fixed-Point Toolbox documentation.

2 Optionally, modify the fixed-point data type definition. See "Editing a Fixed-Point Input Parameter Type" on page 3-17.

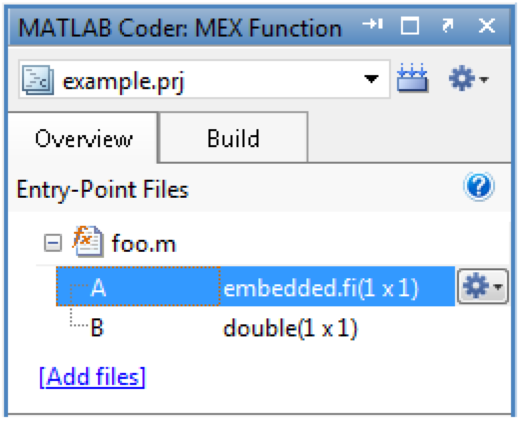
3 Click **OK**.

Editing a Fixed-Point Input Parameter Type

To specify fixed-point inputs, you must install Fixed-Point Toolbox software.

To edit a fixed-point input parameter type:

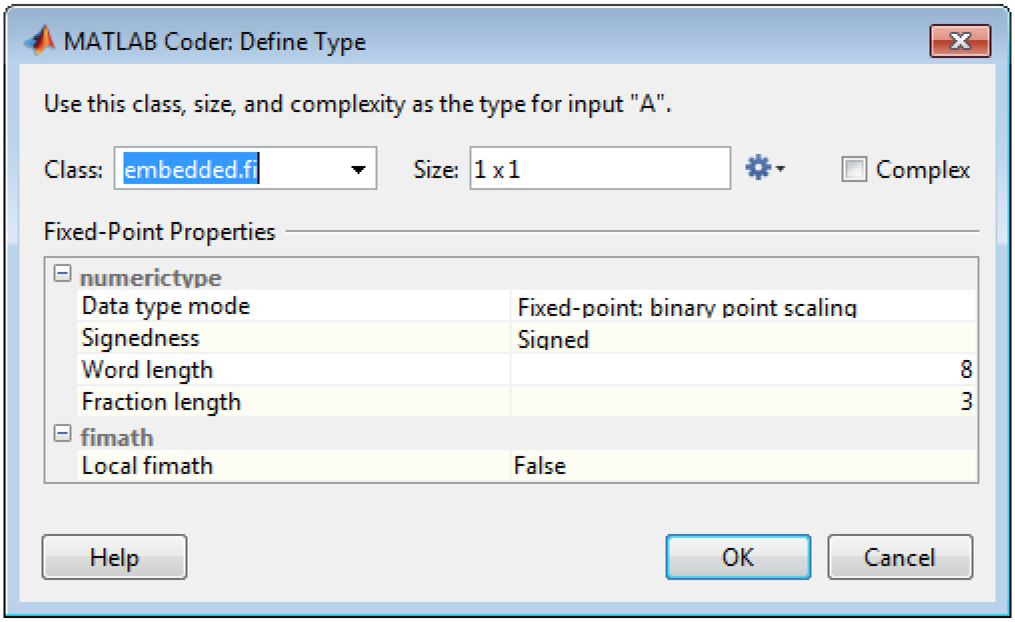
1 On the project **Overview** tab, select the fixed-point input parameter that you want to edit.



2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From this menu, select `Edit Type`.

In the Define Type dialog box, the input type is signed fixed-point with a word length of 8 bits, a fraction length of 3 bits, and uses the global `fimath`.


MATLAB Coder: Define Type

Use this class, size, and complexity as the type for input "A".

Class: embedded.fi   Size: 1 x 1   Complex

Fixed-Point Properties

numerictype	
Data type mode	Fixed-point: binary point scaling
Signedness	Signed
Word length	8
Fraction length	3
fimath	
Local fimath	False

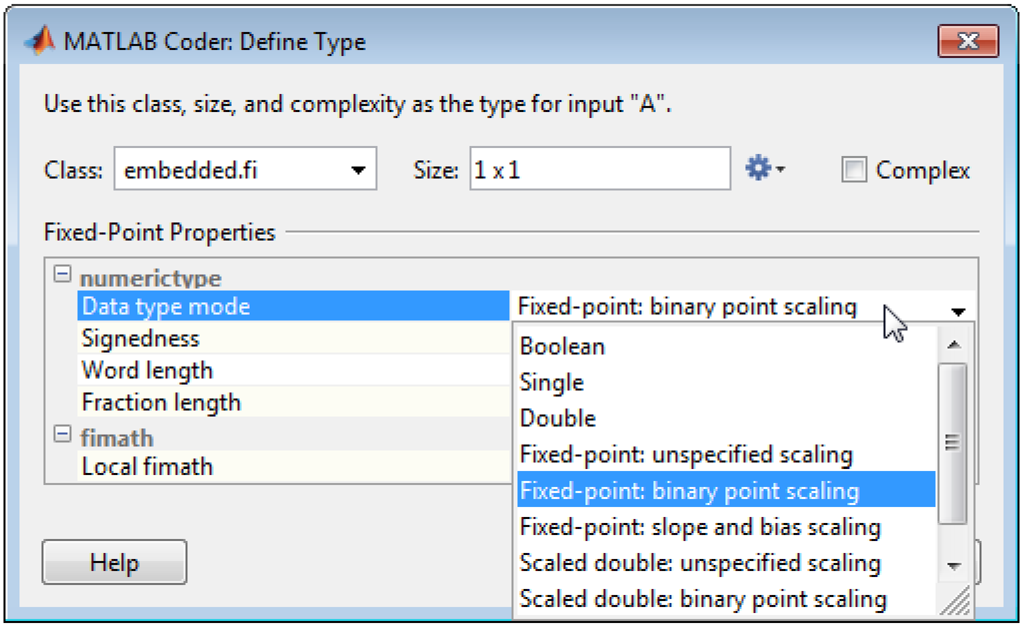
4 Optionally, modify the size and complexity of the input as described in "Defining or Editing Input Parameter Type in a Project" on page 3-12.

5 Modify the numerictype properties, as described in "Modifying numerictype Properties" on page 3-19.

6 Modify the fimath properties, as described in "Modifying fimath Properties" on page 3-20.

Modifying numerictype Properties

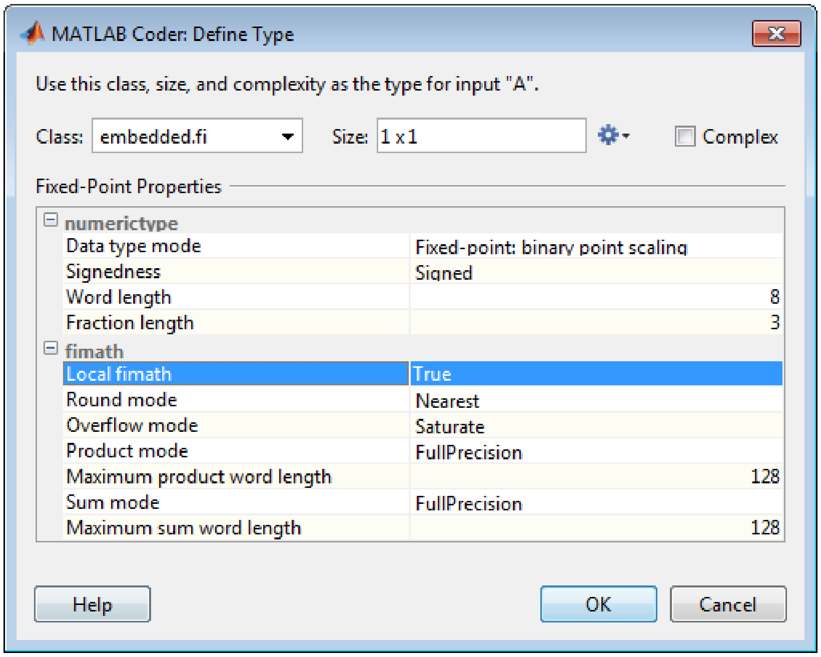
In the Define Type dialog box, under **Fixed-Point Properties**, select the property value that you want to change and set it to the value that you want. For more information, see `numerictype`.


MATLAB Coder: Define Type
Use this class, size, and complexity as the type for input "A".
Class: embedded.fi
Size: 1 x 1
Complex
Fixed-Point Properties
numerictype
Data type mode
Fixed-point: binary point scaling
Signedness
Word length
Fraction length
fimath
Local fimath
Boolean
Single
Double
Fixed-point: unspecified scaling
Fixed-point: binary point scaling
Fixed-point: slope and bias scaling
Scaled double: unspecified scaling
Scaled double: binary point scaling
Help

Modifying fimath Properties

1 In the Define Type dialog box, under **Fixed-Point Properties**, set **Local fimath** to `True`.

   Additional `fimath` properties are displayed.



Use this class, size, and complexity as the type for input "A".

Class: embedded.fi   Size: 1 x 1   Complex [ ]

Fixed-Point Properties

numerictype	
Data type mode	Fixed-point: binary point scaling
Signedness	Signed
Word length	8
Fraction length	3
fimath	
Local fimath	True
Round mode	Nearest
Overflow mode	Saturate
Product mode	FullPrecision
Maximum product word length	128
Sum mode	FullPrecision
Maximum sum word length	128

2 Select the property value that you want to change and set it to the value that you want. For more information, see `fimath`.

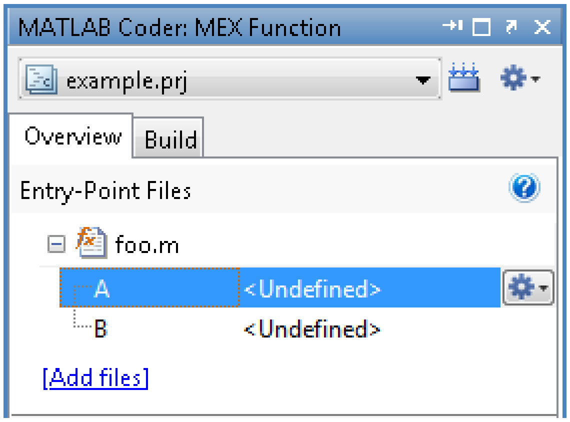
Specifying Structures

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 project **Overview** tab, select the input parameter that you want to define.



2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From the menu, select `Define Type` or `Edit Type`, as applicable.

4 In the Define Type dialog box:

   a From the **Class** drop-down list, select `struct`.

   b Set the size and complexity of the structure as described in "How to Define or Edit an Input Parameter Type" on page 3-12.

   c Click **OK**.

   By default, MATLAB Coder creates a structure containing one field named `field` that has unspecified type.

5 Rename this field as described in "How to Rename a Field in a Structure" on page 3-22. Set its size and complexity as described in "How to Define or Edit an Input Parameter Type" on page 3-12.

6 Optionally, add fields to the structure as described in "How to Add a Field to a Structure" on page 3-23 and then set their size and complexity.

How to Rename a Field in a Structure

1 On the project **Overview** tab, select the structure that you want to rename.

2 To the right of the structure, click the **Actions** icon (⚙) to open the context menu.

3 From the menu, select **Rename**.

4 In the Rename Field dialog box **New Name** field, enter the field name and then click **OK**.

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**.

4 In the Rename Field dialog box **New Name** field, enter the field name and then click **OK**.

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**.

4 In the Rename Field dialog box **New Name** field, enter the field name and then click **OK**.

How to Remove a Field from a Structure

Note You can remove a field from a structure only if it contains more than one field.

---

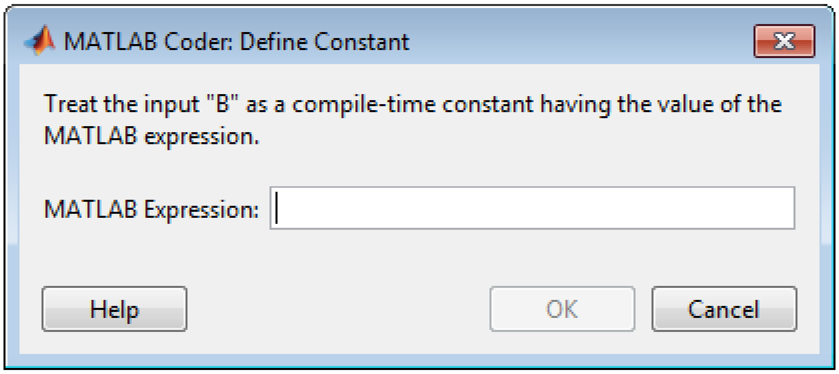
1. In the project **Overview** tab, select the field that you want to remove.
2. To the right of the structure, click the **Actions** icon (⚙) to open the context menu.
3. From the menu, select **Remove Field**.

Defining Constant Input Parameters in a Project

1 On the project **Overview** tab, select the input parameter that you want to define.

2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From this menu, select `Define Constant`.



MATLAB Coder: Define Constant

Treat the input "B" as a compile-time constant having the value of the MATLAB expression.

MATLAB Expression:

4 In the Define Constant dialog box, 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.

Adding Global Variables in a Project

If your MATLAB code uses global variables, add them to the project:

1. On the project **Overview** tab, click **Add global**.
2. In the Rename Global dialog box **New Name** field, enter the name of the global variable and then click **OK**.

   By default, MATLAB Coder names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, etc.
3. After adding a global variable, before building the project, specify its type and, optionally, initial value. If you do not do this, you must create a variable with the same name in the global workspace.

Specifying Global Variable Type and Initial Value in a Project

In this section...

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. Optionally, 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 correct class, size, complexity, and value in the MATLAB workspace.

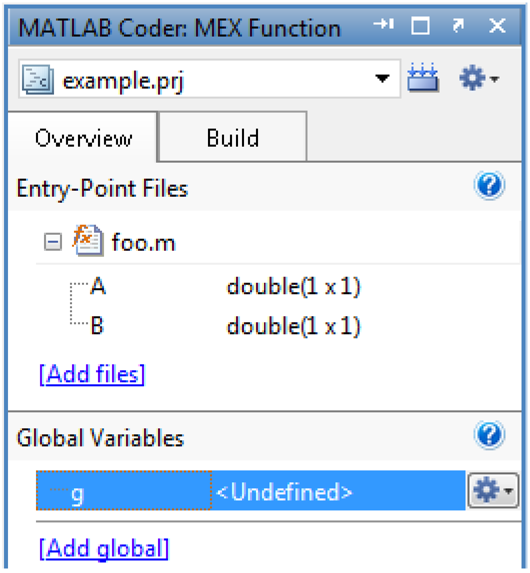
Why Specify a Type Definition for Global Variables?

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.

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 6-82.

Defining a Global Variable by Example

1 On the project **Overview** tab, select the global variable that you want to define.


MATLAB Coder: MEX Function

example.prj

Overview | Build

Entry-Point Files

foo.m

A	double(1 x 1)
B	double(1 x 1)

[Add files]

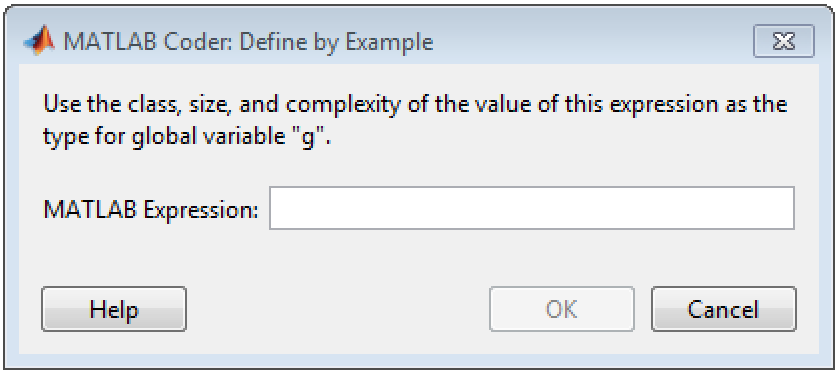
Global Variables

g	<Undefined>

[Add global]

2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From this menu, select `Define by Example`.


MATLAB Coder: Define by Example

Use the class, size, and complexity of the value of this expression as the type for global variable "g".

MATLAB Expression:

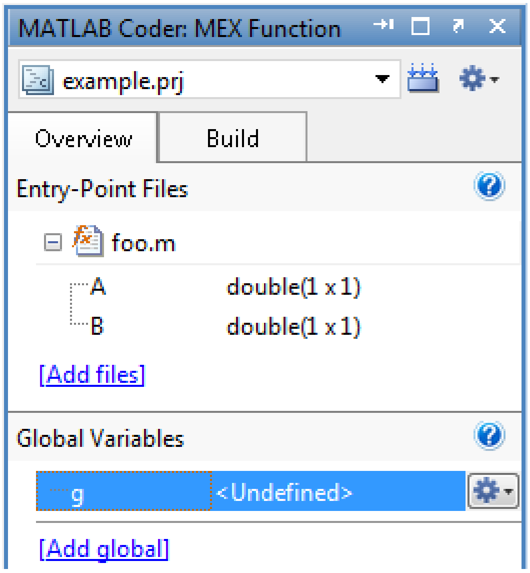
Help | OK | Cancel

4 In the Define by Example dialog box, enter either 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.

Note You define global variables in the same way that you define input parameters. For more information, see "Defining Input Parameters by Example in a Project" on page 3-8

Defining or Editing Global Variable Type

1 On the project **Overview** tab, select the global variable that you want to define.



2 To the right of the parameter, click the **Actions** icon (⚙) to open the context menu.

3 From the menu, select `Define Type` or `Edit Type`, as applicable.

4. In the Define Type dialog box, from the **Class** drop-down list, select a class. The default class is `double`.

   Note If you specify an enumerated type, enter its name in the **Class** field. Define the enumerated type in a file on the MATLAB path. For more information, see "Code Generation for Enumerated Types" on page 6-98.

   ---

5. In the **Size** field, enter the size. The default size is scalar, `1x1`.

6. Optionally, to edit the type definition further, to the right of the **Size** field, click the **Actions** icon.

   MATLAB Coder displays the following menu options:

   - Edit Size Vector Definition

     Select this option to define the size of each dimension of the input. For more information, see "Edit Size Vector Definition" on page 3-14.

   - Make Sizes Variable

     For nonscalar inputs, select this option to make all the dimensions variable size.

     Note For any dimension with a size of `0` or `1`, selecting **Make Sizes Variable** has no effect. However, you can manually make a dimension with a size of `1` variable-size by selecting **Edit Size Vector Definition** and entering a size of `:1` for this dimension.

   - Make Sizes Fixed

     If some or all the dimensions of an input are variable size, select this option to make all the dimensions, except those that are unbounded, fixed in size.

7. Optionally, for numeric types, select **Complex** to make the parameter a complex type. By default, inputs are real.

Defining Global Variable Initial Value

1 On the project **Overview** tab, select the global variable for which you want to specify an initial value.

2 To the right of the variable, click the **Actions** icon (⚙) to open the context menu.

3 From the menu, select `Define Initial Value`.

Note This option is available only if the global variable has a specified type.

---

4 In the Define Initial Value dialog box, enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

Renaming Global Variables

1 On the project **Overview** tab, select the global variable that you want to rename.

2 To the right of the variable, click the **Actions** icon (⚙) to open the context menu.

3 From this menu, select **Rename**.

4 In the Rename Global dialog box **New Name** field, enter the name of the global variable.

Inserting Global Variables

1 On the project **Overview** tab, select the global variable under which you want to insert another global variable.

2 To the right of the variable, click the **Actions** icon (⚙) to open the context menu.

3 From this menu, select **Insert Global**.

4 In the Rename Global dialog box **New Name** field, enter the name of the global variable.

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.

Output File Name Specification

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 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
  - `exe` for C/C++ executables

To learn how to change the default output folder, see "Specifying Output File Locations" on page 3-34.

Command Line Alternative

Use the `codegen` function `-o` option.

Specifying Output File Locations

MathWorks recommends that the path 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
     - `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)
- C/C++ Static 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` and `C/C++ Static Library` or `C/C++ Executable`, you should verify that these settings are correct.

If you enable any of the following parameters that are available for all output types when the output type is `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` and `C/C++ Executable`.

Check These MATLAB Coder Project Parameters When Changing Output Type

Project Settings Dialog Box Tab	Parameter Name
General	Language
	Saturate on integer overflow
	Generated file partitioning method
	Enable variable-sizing
	Dynamic memory allocation

Project Settings Dialog Box Tab	Parameter Name
Paths	Working folder
	Build folder
	Search paths
Report	Always create a code generation report
	Launch report automatically
Comments	Include comments
	MATLAB source code as comments
Symbols	Reserved names
Custom Code	Source file
	Header file
	Initialize function
	Terminate function
	Include directories
	Source files
	Libraries
	Post-code-generation command
Optimization	Use memcpy for vector assignment
	Memcpy threshold (bytes)
	Use memset to initialize floats and doubles to 0.0
Advanced	Inline threshold
	Inline threshold max
	Inline stack limit
	Stack usage max
	Constant folding timeout

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

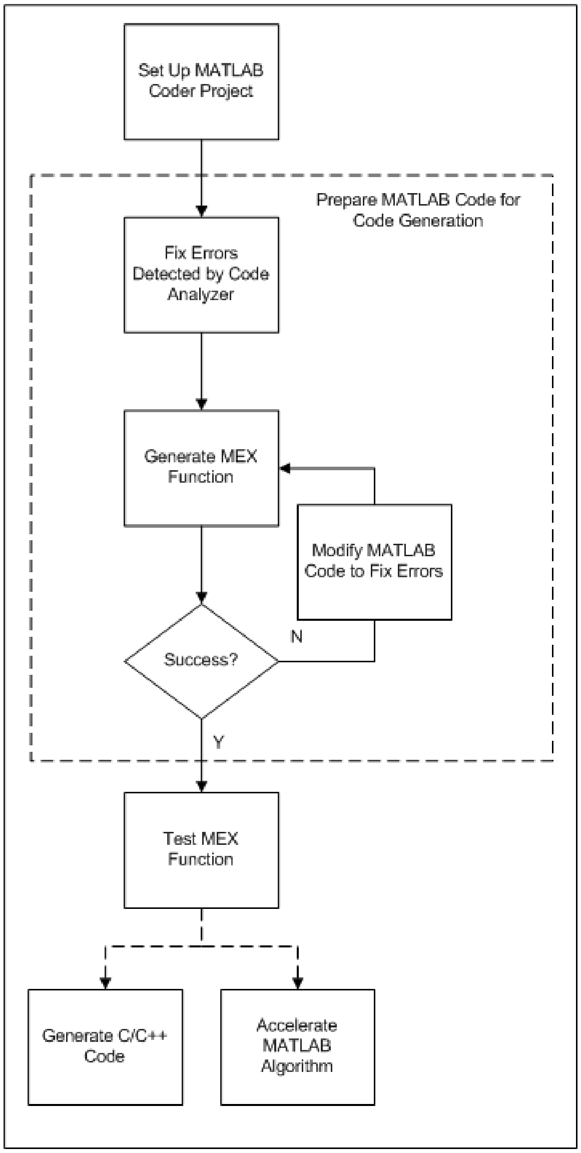
- ReservedNameArray
- SaturateOnIntegerOverflow
- StackUsageMax
- TargetLang

More About

- “Primary Function Input Specification” on page 6-30
- “How Working with Variable-Size Data Is Different for Code Generation”
- “Code Generation for Complex Data”
- “Code Generation for Enumerated Types” on page 6-98
- “Generating C Code from MATLAB Code Using the MATLAB Coder Project Interface”

Preparing MATLAB Code for C/C++ Code Generation

Preparing MATLAB Code for C/C++ Code Generation Workflow


Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Fix Errors Detected by Code Analyzer
Generate MEX Function
Modify MATLAB Code to Fix Errors
N
Success?
Y
Test MEX Function
Generate C/C++ Code
Accelerate MATLAB Algorithm

See Also

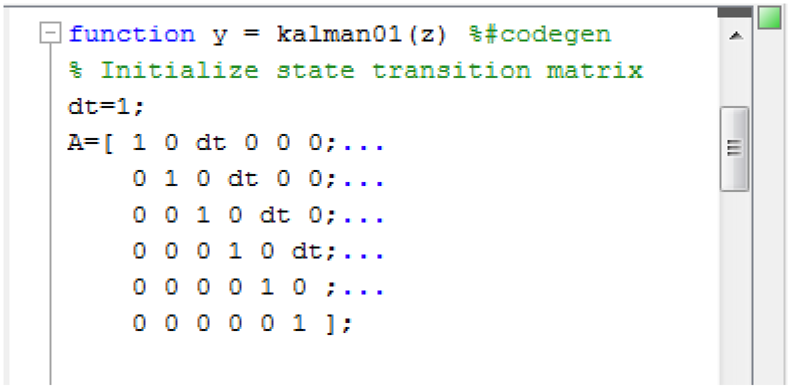
- Chapter 3, "Setting Up a MATLAB® Coder Project"
- "Fixing Errors Detected by the Code Analyzer" on page 4-4
- "How to Generate MEX Functions Using the MATLAB® Coder Project Interface" on page 4-6
- "Fixing Errors Detected at Code Generation Time" on page 4-15
- Chapter 5, "Testing MEX Functions in MATLAB"
- Chapter 6, "Generating C/C++ Code from MATLAB Code"
- Chapter 7, "Accelerating MATLAB Algorithms"

Fixing Errors Detected by the Code Analyzer

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing code generation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications to maximize performance and maintainability. The analyzer indicator is in the top-right corner of the MATLAB Editor window. If the code analyzer detects no issues, it is green. It turns orange to indicate warnings and red to indicate errors.

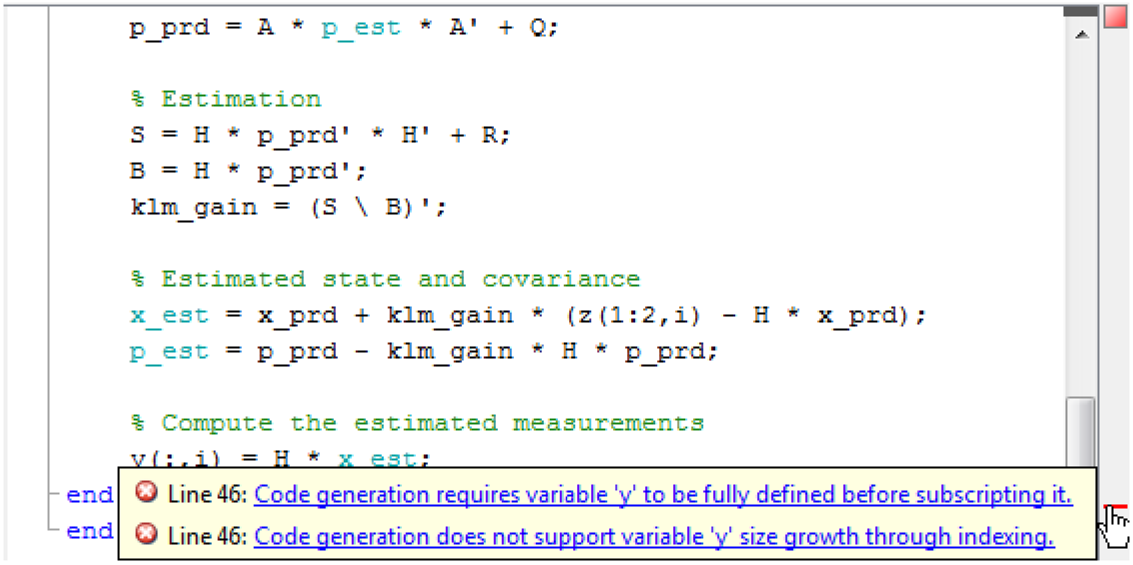
To use the code analyzer to identify warnings and errors specific to code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of code generation messages is available in the MATLAB Code Analyzer preferences. For more information, see “Using the MATLAB Code Analyzer Report” in the MATLAB Desktop Tools and Development Environment documentation.

Here, the code analyzer message indicator is green, indicating that it has not detected any 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, a red 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
    y(:,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 any errors detected by the code analyzer . For more information about how to fix these errors, see “Changing Code Based on Code Analyzer Messages” in the MATLAB Desktop Tools and Development Environment documentation.

See Also

- “Design Considerations When Writing MATLAB Code for Code Generation” on page 4-16
- “Debugging Strategies” on page 4-18

How to Generate MEX Functions Using the MATLAB Coder Project Interface

Workflow

Step	Action	Details
1	Set up your MATLAB Coder project.	Chapter 3, “Setting Up a MATLAB® Coder Project”
2	Fix any errors detected by the code analyzer.	“Fixing Errors Detected by the Code Analyzer” on page 4-4
3	Select whether to generate code only.	“Generating Code Only” on page 4-9
4	Specify build configuration parameters.	“Configuring Project Settings” on page 4-10
5	Build the project.	“Building a MATLAB® Coder Project” on page 4-11

Generating a MEX Function 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:

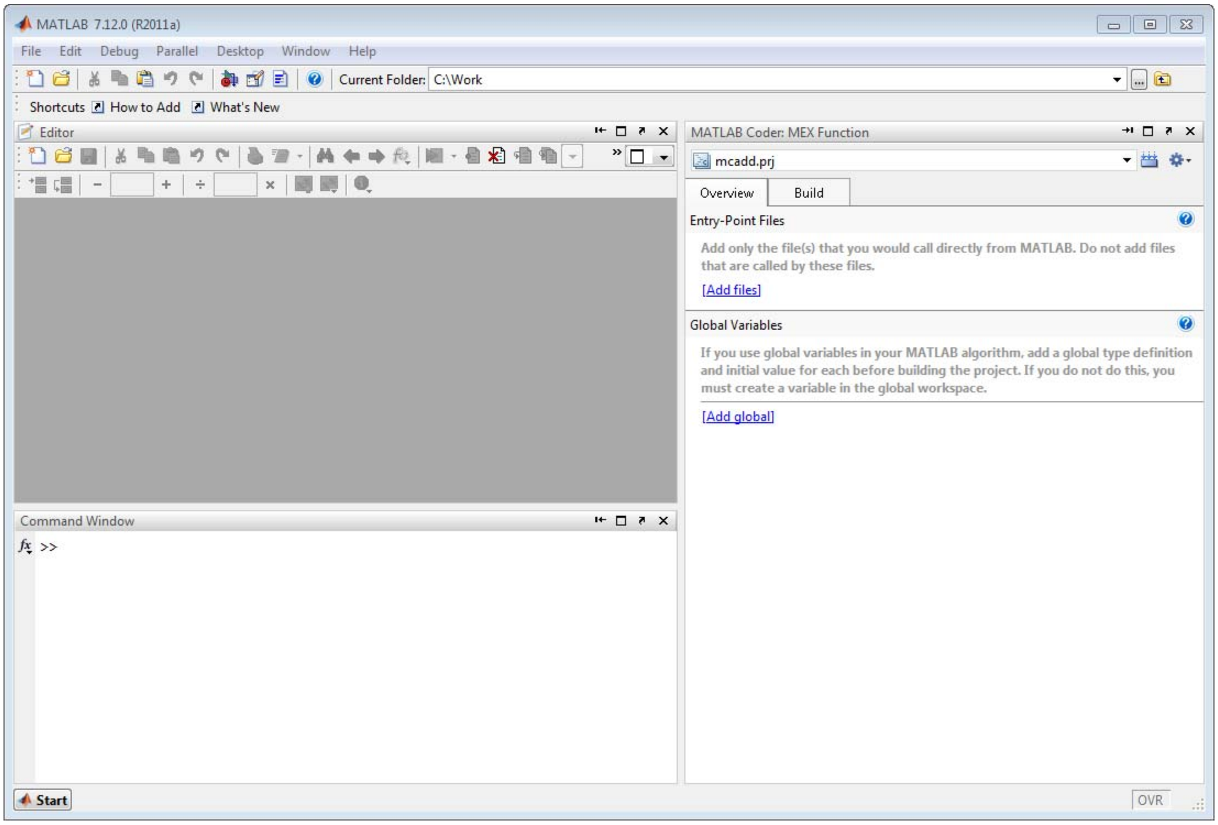
```
function y = mcadd(u,v) %#codegen
y = u + v;
```

2 In the same folder, set up a MATLAB Coder project.

   a At the MATLAB command line, enter

   ```
   coder -new mcadd.prj
   ```

   By default, the project opens in the MATLAB workspace on the right side.

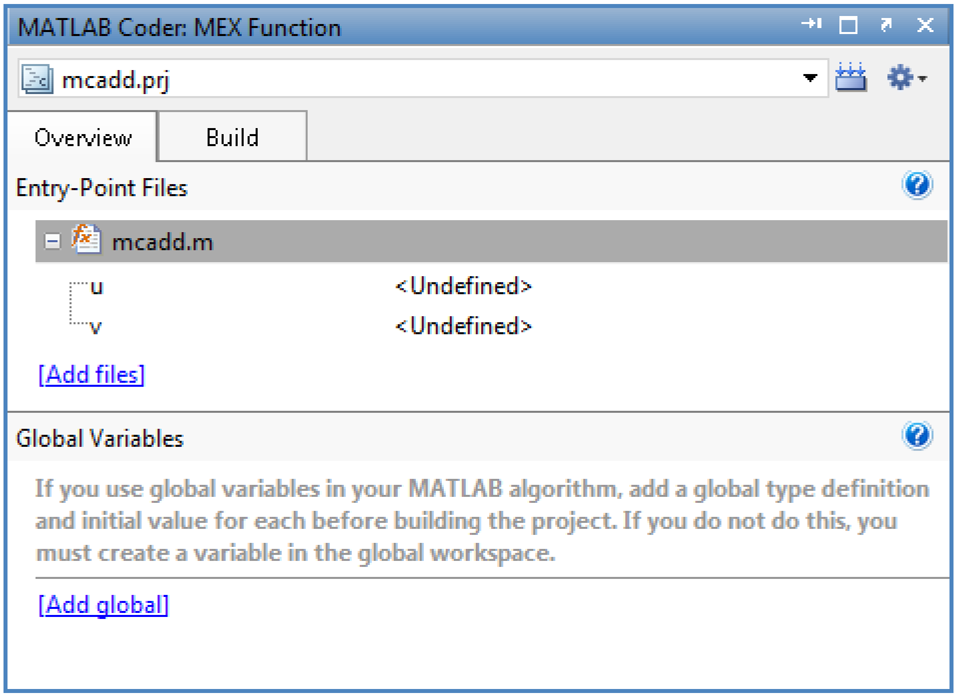


Add only the file(s) that you would call directly from MATLAB. Do not add files that are called by these files.

If you use global variables in your MATLAB algorithm, add a global type definition and initial value for each before building the project. If you do not do this, you must create a variable in the global workspace.

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.



u	<Undefined>
v	<Undefined>

If you use global variables in your MATLAB algorithm, add a global type definition and initial value for each before building the project. If you do not do this, you must create a variable in the global workspace.

c Define the type of input u.

i On the **Overview** tab, select the input parameter u. To the right of this parameter, click the **Actions** icon (⚙) to open the context menu.

ii From the menu, select `Define Type`.

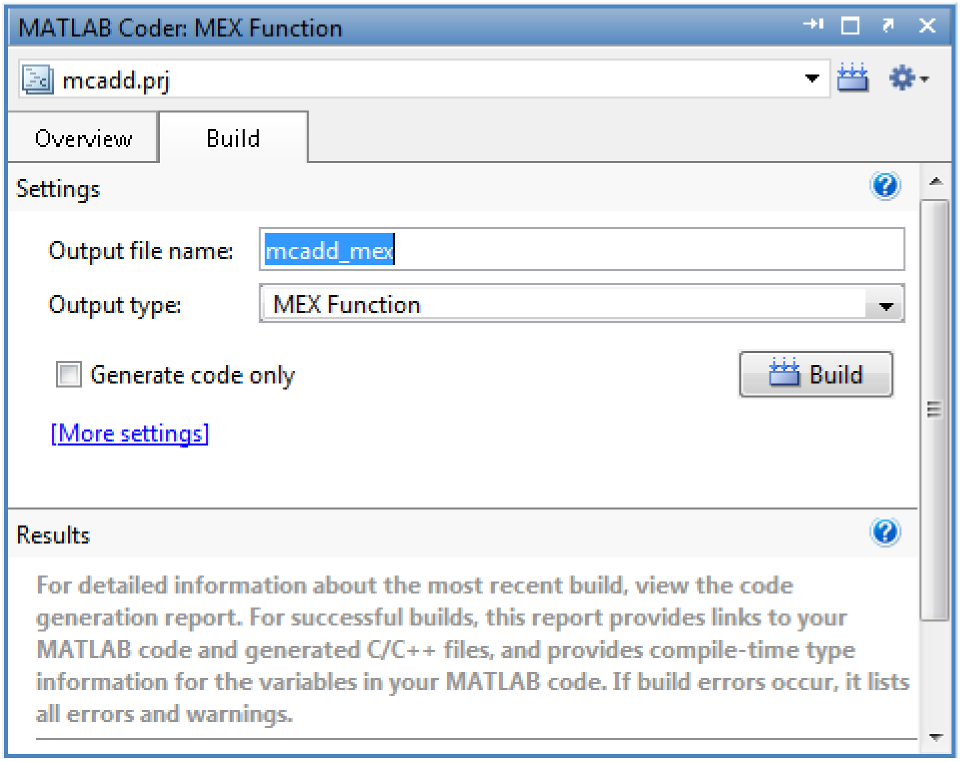
iii In the **Define Type** dialog box, set **Class** to `int16`. Click **OK** to specify that the input is a scalar `int16`.

MATLAB Coder displays scalars with a size `1x1`.

d Repeat step c for 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`.



Output file name: mcadd_mex

Output type: MEX Function

Generate code only

For detailed information about the most recent build, view the code generation report. For successful builds, this report provides links to your MATLAB code and generated C/C++ files, and provides compile-time type information for the variables in your MATLAB code. If build errors occur, it lists all errors and warnings.

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 6-71.

You can now test your MEX function in MATLAB. For more information, see Chapter 5, "Testing MEX Functions in MATLAB".

Generating Code Only

On the project **Build** tab, select **Generate code only**.

If you want to generate C/C++ code only, select this option. Doing so does not invoke the make command or generate compiled object code.

When to Generate Code Only

- After successful MEX generation.
- During the development cycle, when you want to iterate rapidly between modifying MATLAB code and generating code and you want to inspect the code.
- You want to compile the code with your own compiler.

Configuring 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` to `C/C++ Static Library` or `C/C++ Executable`, check these settings to verify that they are correct. For more information, see "Changing Output Type" on page 3-35.

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 in the dialog box.

See Also

- "How to Enable Code Generation Reports in the Project Settings Dialog Box" on page 6-177
- "In the Project Settings Dialog Box" on page 6-120

- "How to Disable Inlining Globally in the Project Settings Dialog Box" on page 6-131
- "Generating Traceable Code" on page 6-89
- "Disabling Run-Time Checks in the Project Settings Dialog Box" on page 7-15
- "Disabling BLAS Library Support in the Project Settings Dialog Box" on page 7-12

Building 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. For successful builds, 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 all 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 6-174
- "Code Generation for More Than One Entry-Point MATLAB Function" on page 6-73

- “Code Generation for Global Data” on page 6-80

See Also

- “Code Generation for More Than One Entry-Point MATLAB Function” on page 6-73
- “Code Generation for Global Data” on page 6-80
- “Output File Name Specification” on page 3-33
- “Specifying Output File Locations” on page 3-34

How to Generate MEX Functions at the Command Line

Workflow for Generating MEX Functions at the Command Line

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 6-70
4	Fix any errors detected by the code analyzer.	"Fixing Errors Detected by the Code Analyzer" on page 4-4
5	Specify build configuration parameters.	"How to Specify Build Configuration Parameters" on page 6-22
6	Specify properties of primary function inputs.	"Primary Function Input Specification" on page 6-30
7	Generate the MEX function using `codegen` with the appropriate command-line options.	"Generating MEX Functions at the Command Line Using codegen" on page 4-14

Generating a MEX Function 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 6-71.

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 “How to Generate MEX Functions at the Command Line” on page 4-13.

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 6-30
- “Generating MEX Functions from MATLAB Code at the Command Line”
- “Code Generation for More Than One Entry-Point MATLAB Function” on page 6-73
- “Code Generation for Global Data” on page 6-80

Fixing Errors Detected at Code Generation Time

When MATLAB Coder 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 “About Code Generation from MATLAB Algorithms” in the MATLAB Coder documentation. MathWorks recommends that you choose a debugging strategy for detecting and correcting code generation errors in your MATLAB code. For more information, see “Debugging Strategies” on page 4-18.

After successful code generation, MATLAB Coder 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 you declare these functions to be extrinsic, MATLAB Coder attempts to compile these functions. See “How MATLAB Resolves Function Calls in Generated Code” in the Code Generation from MATLAB documentation. 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 6-174
- “Why Test MEX Functions in MATLAB?” on page 5-4
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 4-16
- “Debugging Strategies” on page 4-18
- “Declaring MATLAB Functions as Extrinsic Functions”

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 performance, 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 successfully 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

- “About Code Generation from MATLAB Algorithms” in the Code Generation from MATLAB documentation
- “Defining MATLAB Variables for C/C++ Code Generation”in the Code Generation from MATLAB documentation
- “How Working with Variable-Size Data Is Different for Code Generation” in the Code Generation from MATLAB documentation
- “Bounded Versus Unbounded Variable-Size Data” in the Code Generation from MATLAB documentation
- “Enabling and Disabling Dynamic Memory Allocation” on page 6-100
- “Controlling Run-Time Checks” on page 7-14

Debugging Strategies

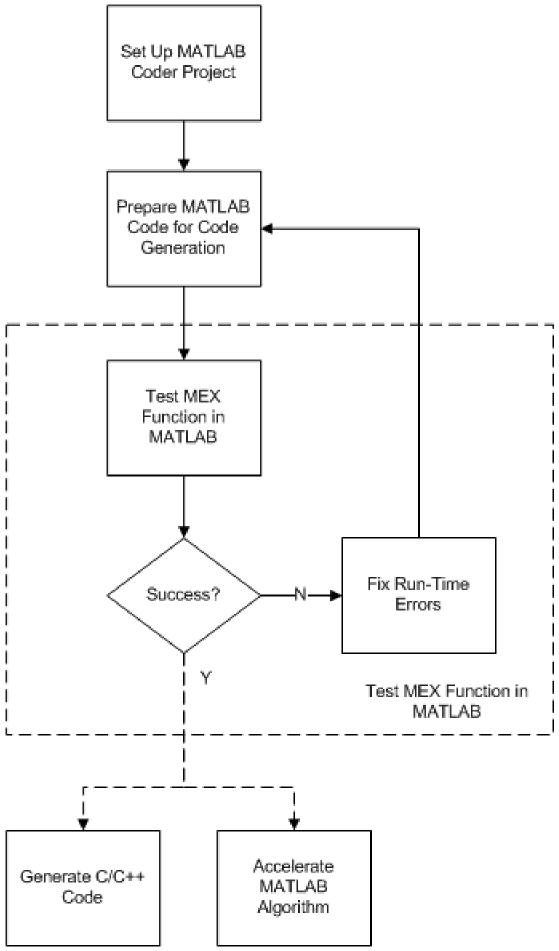
Before you perform code verification, MathWorks recommends that you choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other's functions. The following table describes two general strategies, each of which has advantages and disadvantages.

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	**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.	• 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	**1** Declare all 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" in the Code Generation from MATLAB documentation. **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: • Extrinsic declarations • Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values (see "Working with mxArrays" in the Code Generation from MATLAB documentation).

Testing MEX Functions in MATLAB

Workflow for Testing MEX Functions in MATLAB


Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Test MEX Function in MATLAB
Success?
N
Fix Run-Time Errors
Y
Test MEX Function in MATLAB
Generate C/C++ Code
Accelerate MATLAB Algorithm

See Also

- Chapter 3, “Setting Up a MATLAB® Coder Project”

- Chapter 4, “Preparing MATLAB Code for C/C++ Code Generation”
- “Why Test MEX Functions in MATLAB?” on page 5-4
- “Debugging Run-Time Errors” on page 5-5
- Chapter 6, “Generating C/C++ Code from MATLAB Code”
- Chapter 7, “Accelerating MATLAB Algorithms”

Why Test MEX Functions in MATLAB?

Before generating C/C++ code for your MATLAB code, it is best practice to test the MEX function to verify that it provides the same functionality as the original MATLAB code. To do this, run the MEX function using the same inputs as you used to run the original MATLAB code and compare the results. For example, see "Validating the MEX Function" in the Generating MEX Functions from MATLAB Code at the Command Line tutorial.

In addition, running the MEX function in MATLAB before generating code enables you to detect and fix run-time errors that would be much harder to diagnose in the generated code. If you encounter run-time errors in your MATLAB functions, you should fix them before generating code. For more information, see "Debugging Run-Time Errors" on page 5-5.

By default, the following run-time checks execute when you run your MEX function in MATLAB:

- Memory integrity checks. These checks perform array bounds and dimension checking and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution with 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** at any time.

For more information, see "Controlling Run-Time Checks" on page 7-14.

Debugging Run-Time Errors

In this section...

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 5-5"Handling Run-Time Errors" on page 5-7.

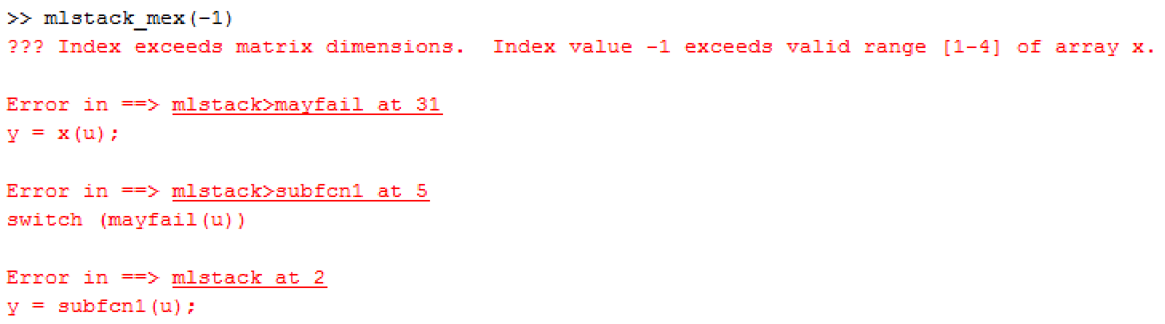
Viewing Errors in the Run-Time Stack

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 at 31
y = x(u);

Error in ==> mlstack>subfcn1 at 5
switch (mayfail(u))

Error in ==> mlstack at 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 performance. Consider disabling the run-time stack for faster performance.

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 7-15.

Caution Before disabling the memory integrity checks, you should verify that all array bounds and dimension checking is unnecessary.

Handling Run-Time Errors

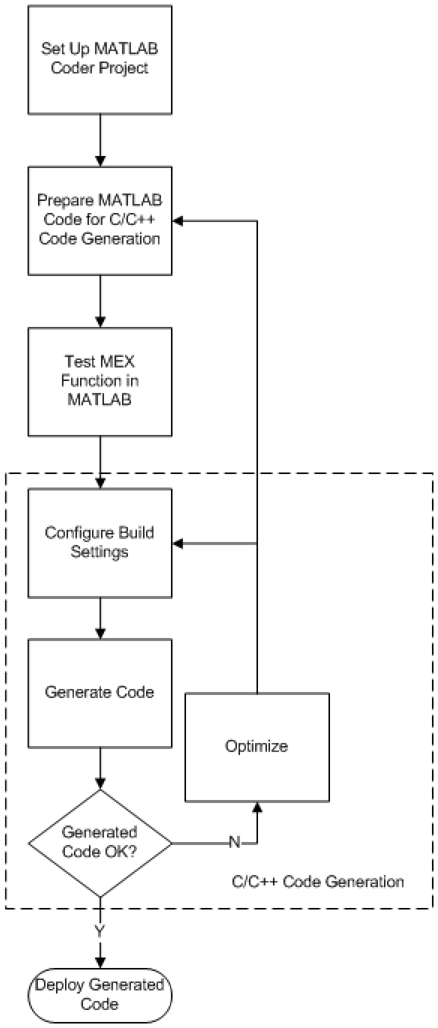
The code generation software propagates error ID's. If you throw an error or warning in your MATLAB code, use the `try-catch` statement in your test bench code to examine the error information and attempt to recover, or clean up and abort. For example, for the function in "Example Run-Time Stack Trace" on page 5-6, 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" in the *MATLAB Programming Fundamentals* documentation.

Generating C/C++ Code from MATLAB Code

Code Generation Workflow



See Also

- Chapter 3, “Setting Up a MATLAB® Coder Project”
- Chapter 4, “Preparing MATLAB Code for C/C++ Code Generation”
- Chapter 5, “Testing MEX Functions in MATLAB”
- “Build Setting Configuration” on page 6-15
- “C/C++ Code Generation” on page 6-5
- “Code Optimization” on page 6-64

C/C++ Code Generation

Using MATLAB Coder, you can generate standalone C/C++ static libraries and C/C++ executables. By default, MATLAB Coder, if no code generation errors occur, MATLAB Coder 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 Static C/C++ Libraries from MATLAB Code" on page 6-6
C/C++ executables from your MATLAB code	"Standalone C/C++ Executables from MATLAB Code" on page 6-8
MEX functions from your MATLAB code	"How to Generate MEX Functions Using the MATLAB® Coder Project Interface" on page 4-6

If there are errors, 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 6-174.

Specifying 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
`.o` , `.obj`	Custom object file
`.a` , `.lib`, `.so`	Library
`.tmf`	Template makefile for custom MATLAB Coder builds

Generating Static C/C++ Libraries from MATLAB Code

In this section...

Generating a C Static Library Using the Project Interface

In this example, you create a MATLAB function that adds two numbers and 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 static C 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.

c Define the type of input `u`.

i On the **Overview** tab, select the input parameter `u`. To the right of this parameter, click the Actions icon (⚙) to open the context menu.

ii From the menu, select `Define Type`.

  iii In the **Define Type** dialog box, set **Class** to `int16`. Click **OK** to specify that the input is a scalar `int16`.

 d Repeat step c for input `v`.

3 In the MATLAB Coder project, click the **Build** tab.

4 On this tab, set the **Output type** to `C/C++ Static library`.

  The default output file name is `mcadd`.

5 Click the **Build** button to generate a library using the default project settings.

  MATLAB Coder builds the project and generates a C library and supporting files in the default folder, `codegen/lib/mcadd`.

Generating a C Static Library at the Command Line Using codegen

In this example, you create a MATLAB function that adds two numbers. You generate a static C library for this function, specifying types for the function input parameters.

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 . 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 library with the default name, `mcadd`, and supporting files in the default folder, `codegen/lib/mcadd`.

Standalone C/C++ Executables from MATLAB Code

In this section...

Generating 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 6-12.

---

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`.

See Also

- Chapter 3, "Setting Up a MATLAB® Coder Project"
- Chapter 4, "Preparing MATLAB Code for C/C++ Code Generation"
- Chapter 5, "Testing MEX Functions in MATLAB"
- "Build Setting Configuration" on page 6-15
- "C/C++ Code Generation" on page 6-5
- "Code Optimization" on page 6-64

Generating 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`.

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 6-52.

How to 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, under **Additional files and directories to be built**, set:

   a **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 6-12.

   b **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 6-24).

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 6-12.) 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 appropriate command line options. For example, if `myFunction` takes one input parameter of type `double`:

   ```
   codegen -config cfg  myMFunction -args {0}
   ```

   MATLAB Coder compiles and links the main function with the C/C++ code that it generates from `myMFunction.m`.

Specifying main Functions in the Code Generation Dialog Box

1. Create a configuration object for an executable;

   ```
   cfg = coder.config('exe');
   ```

2. Open the Code Generation dialog box:

   ```
   open cfg
   ```

3. On the **Custom Source** tab, under **Include list of additional**:

   a. Set **Source files** to the name of the C/C++ source file that contains the `main` function. For example, `main.c`.

   b. Set **Include directories** to the location of `main.c`. For example, `c:\myfiles`.
   c. Click **Apply**.

4. Generate the C/C++ executable using the appropriate command line options. For example, if `myFunction` takes one input parameter of type `double`:

   ```
   codegen -config cfg  myMFunction -args {0}
   ```

   MATLAB Coder compiles and links the main function with the C/C++ code that it generates from `myMFunction.m`.

Build Setting Configuration

In this section...

Output Type Specification

Output Types

MATLAB Coder can generate code for the following output types:

- MEX function
- Standalone C/C++ code and compile it to a 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 6-12.

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 6-127.

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, from the **Output type** drop-down list, select one of the available output types:

- MEX Function (default)
- C/C++ Static 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` and `C/C++ Static Library` or `C/C++ Executable`, verify that these settings are correct. For more information, see "Changing Output Type" on page 3-35.

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 6-30.

Note C is the default language for code generation with MATLAB Coder. To generate C++ code, see "Specifying a Language for Code Generation" on page 6-18.

To Generate:	Use This Command:
MEX function using the default code generation options	`codegen foo`

MEX function specifying code generation options	`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`

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	`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`

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	`codegen -config:exe main.c foo` **Note** You must specify a `main` function for generating a C/C++ executable. See "Specifying main Functions for C/C++ Executables" on page 6-12

To Generate:	Use This Command:

Standalone C/C++ code and compile it to an executable specifying code generation options	`cfg = coder.config('exe');`
	`% Set configuration parameters, for example,`
	`%  specify main file`
	`cfg.CustomSource = 'main.c';`
	`cfg.CustomInclude = 'c:\myfiles';`
	`codegen -config cfg foo`
	
	Note You must specify a `main` function for generating a C/C++ executable. See "Specifying main Functions for C/C++ Executables" on page 6-12

Specifying a Language for Code Generation

- "Specifying a Language for Code Generation in the Project Settings Dialog Box" on page 6-18
- "Specifying a Language for Code Generation at the Command Line" on page 6-19
- "Specifying a Language for Code Generation Using Configuration Object Dialog Boxes" on page 6-19

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, the command line, or configuration object dialog boxes.

Specifying a Language for Code Generation in the Project Settings Dialog Box

1. Select the appropriate 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 **General** tab, set **Language** to `C` or `C++`.

See Also.

- “Setting Up the C/C++ Compiler”
- Chapter 3, “Setting Up a MATLAB® Coder Project”

Specifying a Language for Code Generation at the Command Line

1. Select the appropriate 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++';
   ```

See Also.

- “Working with Configuration Objects” on page 6-24
- “Setting Up the C/C++ Compiler”

Specifying a Language for Code Generation Using Configuration Object Dialog Boxes

1. Select the appropriate compiler for your target language.
2. Create a configuration object for code generation. For example, for a library:

   ```
   cfg = coder.config('lib');
   ```

3. Open the configuration object dialog box:

   ```
   open cfg
   ```

4. On the **General** tab, set **Language** to `C` or `C++`.

See Also.

- "Working with Configuration Objects" on page 6-24
- "Setting Up the C/C++ Compiler"

Output File Name Specification

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 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
  - `exe` for C/C++ executables

Command Line Alternative

Use the `codegen` function `-o` option.

Specifying Output File Locations

Specifying Output File Location in a Project

MathWorks recommends that the output file location 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
     - `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 6-22
`codegen` 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 page 6-23
`codegen` in build scripts		
`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 6-28

How to Specify Build Configuration Parameters

- "Specifying Build Configuration Parameters in the Project Settings Dialog Box" on page 6-22
- "Specifying Build Configuration Parameters at the Command Line Using Configuration Objects" on page 6-23
- "Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes" on page 6-28

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` and `C/C++ Static Library` or `C/C++ Executable`, verify that these settings are correct. For more information, see "Changing Output Type" on page 3-35.

---

2. Modify the parameters as necessary. 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
`coder.CodeConfig`	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 `coder.CodeConfig`.
`coder.EmbeddedCodeConfig`	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 `coder.EmbeddedCodeConfig`.

Configuration Object	Description
`coder.HardwareImplementation`	Specifies parameters of the target hardware implementation. If not specified, `codegen` generates code that is compatible with the MATLAB host computer. For more information, see the class reference information for `coder.HardwareImplementation`.
`coder.MexCodeConfig`	Specifies parameters for MEX code generation. For more information, see the class reference information for `coder.MexCodeConfig`.

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 6-26.

   For example, to generate a configuration object for C 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 necessary, using one of these methods:

   - Interactive commands, as described in "Specifying Build Configuration Parameters at the Command Line Using Configuration Objects" on page 6-23
   - Dialog boxes, as described in "Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes" on page 6-28

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 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 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 `coder.MexCodeConfig`	`cfg = coder.config('mex');`

Code generation configuration object for generating a standalone C/C++ library or executable
 	
`coder.CodeConfig`
	`% To generate a static library`
	`cfg = coder.config('lib');`
	`% To generate an executable`
	`cfg = coder.config('exe');`
	
	Note If an Embedded Coder license is available, creates a `coder.EmbeddedCodeConfig` object.
	
	If you use concurrent licenses, to disable check out of an Embedded Coder license, use one of the following commands:
	
	`cfg = coder.config('lib', 'ecoder', false)`
	
	`cfg = coder.config('exe', 'ecoder', false)`

To Create...	Use a Command Such As...

Code generation configuration object for generating a standalone C/C++ library or executable for an embedded target
 	
`coder.EmbeddedCodeConfig`
	`% To generate a static library`
	`cfg = coder.config('lib');`
	`% To generate an executable`
	`cfg = coder.config('exe');`
	
	Note Requires an Embedded Coder license; otherwise creates a `coder.CodeConfig` object.

Hardware implementation configuration object `coder.HardwareImplementation`	`hwcfg = coder.HardwareImplementation`

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 6-27.

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++ 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 6-26.

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 needed, 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`.

Primary Function Input Specification

In this section...

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 subfunctions 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**	**numerictype**	**fimath**
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	**Specify properties for each field according to its class** 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: • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify `numerictype`, and `fimath`.				
All 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`
`numerictype`	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 "Example: Specifying Class and Size of Scalar Structure" on page 6-49 and "Example: Specifying Class and Size of Structure Array" on page 6-50.

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 global `fimath`. The MATLAB factory default has the following properties:

```
            RoundMode: nearest
         OverflowMode: saturate
          ProductMode: FullPrecision
 MaxProductWordLength: 128
              SumMode: FullPrecision
     MaxSumWordLength: 128
```

For more information, see "Working with the Global fimath" in the Fixed-Point Toolbox documentation.

When running MEX functions that depend on the MATLAB default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time error, 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 `text_mex` to use the factory setting of the MATLAB default `fimath`:

```
resetglobalfimath;
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 modify the MATLAB default `fimath` value so it no longer matches the factory setting used at compile time:

```
F = fimath('RoundMode','Floor');
globalfimath(F);
```

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
`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 3-7 **Note** If you define input properties programmatically in the MATLAB file, you cannot use this method	• If you are working in a MATLAB Coder project, easy to use • Does not alter original MATLAB code • MATLAB Coder saves the definitions in the project file	• Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Defining Input Properties by Example at the Command Line" on page 6-36 **Note** If you define input properties programmatically in the MATLAB file, you cannot use this method	• Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a small number of primary inputs	• Must be specified at the command line every time you invoke `codegen` (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Defining Input Properties Programmatically in the MATLAB File" on page 6-42	• Integrated with MATLAB code; no need to redefine properties each time you invoke MATLAB Coder • Provides documentation of property specifications in the MATLAB code	• Uses complex syntax • MATLAB Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must

Method	Advantages	Disadvantages
	• Efficient for specifying memory-intensive inputs such as large structures	reenter the input types in the project.

Defining Input Properties by Example at the Command Line

- "Command Line Option -args" on page 6-36
- "Rules for Using the -args Option" on page 6-36
- "Specifying Constant Inputs at the Command Line" on page 6-37
- "Specifying Variable-Size Inputs at the Command Line" on page 6-38
- "Example: Specifying Properties of Primary Inputs by Example at the Command Line" on page 6-40
- "Example: Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line" on page 6-41

Command Line Option -args

The `codegen` function provides a command-line option `-args` for specifying the properties of primary 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 file with `codegen`.

See "Example: Specifying General Properties of Primary Inputs" on page 6-48 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 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.

Example: 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)}
```

Specifying 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 "Code Generation for Variable-Size Data" on page 6-99.

Example: 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}
```

Example: 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
  ```

Example: 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 Toolbox 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;
```

Defining 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.

- "How to Use assert with MATLAB® Coder" on page 6-42
- "Rules for Using assert Function" on page 6-47
- "Example: Specifying General Properties of Primary Inputs" on page 6-48
- "Example: Specifying Properties of Primary Fixed-Point Inputs" on page 6-48
- "Example: Specifying Class and Size of Scalar Structure" on page 6-49
- "Example: Specifying Class and Size of Structure Array" on page 6-50

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 6-43
- "Specify fi Class" on page 6-43
- "Specify Structure Class" on page 6-44
- "Specify Any Size" on page 6-44
- "Specify Scalar Size" on page 6-45
- "Specify Real Input" on page 6-45

- "Specify Complex Input" on page 6-45
- "Specify numerictype of Fixed-Point Input" on page 6-46
- "Specify fimath of Fixed-Point Input" on page 6-46
- "Specify Multiple Properties of Input" on page 6-46

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 6-46. You can also set its fimath properties, see "Specify fimath of Fixed-Point Input" on page 6-46. 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 6-46. You can also set its `fimath` properties, see "Specify fimath of Fixed-Point Input" on page 6-46. 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 Any 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 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 = numerictype(1, 32, 30);

% Set the numerictype property of input U to T.
assert(isequal(numerictype(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 any 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 6-46. You can also set its `fimath` properties. See "Specify fimath of Fixed-Point Input" on page 6-46. 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.

Example: 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]));
...
```

Example: Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Toolbox 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
class	`fi`
`numerictype`	`numerictype` object `T`, as specified in the primary function
`fimath`	`fimath` object `F`, as specified in the primary function
size	`scalar`
complexity	`real` (by default)

```
function y = mcsqrtfi(x) %#codegen
T = numerictype('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(numerictype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Example: 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:

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

% 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

Example: 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.

Calling Generated C/C++ Functions

In this section...

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 "Eliminating Redundant Copies of Function Inputs (A=foo(A))" on page 6-67.

Calling 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. 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);
```

Here, the MATLAB function `callmyCFunction` calls the custom C function `myCFunction`, which takes one input argument.

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 6-52.
- 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 6-56.
- 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 6-56.

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
primary_function_name`_initialize`	Before you call your C/C++ executable or library function for the first time
primary_function_name`_terminate`	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.

Example: Calling a C/C++ 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);
```

To generate a C 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 library function.

     The default target language is C. To change the target language to C++, see "Specifying a Language for Code Generation" on page 6-18.

   - MATLAB Coder creates the library `absval.lib` (or `absval.a` on Linus 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 "Defining Input Properties by Example at the Command Line" on page 6-36.

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 "Calling Generated C/C++ Functions" on page 6-51.

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/absva
     ```

     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
     ```

   b At the MATLAB prompt, call the C library by running the MEX function. For example, on Windows:

   ```
   callabsval_mex
   ```

Example: Calling a C Library Function from C Code

This example shows how to call a generated C library function from C code. It uses the C library function `absval` described in "Example: Calling a C/C++ Library Function from MATLAB Code" on page 6-53.

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 6-58.

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)
   ```

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

- "Calling Generated C/C++ Functions" on page 6-51
- `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

- "Calling Generated C/C++ Functions" on page 6-51
- `coder.wref` function reference information
- `coder.ceval` function reference information

Custom C/C++ Code Integration

In this section...

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**, set **Source file** and **Header file** as needed. **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.

Custom Code Property	Description
Source files	Specifies additional custom C/C++ files to be compiled with the MATLAB file.
Libraries	Specifies the names of object or library files to be linked with the generated code.
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 6-26.

   For example:

   ```
   cc = coder.config('lib');
   ```

2 Set one or more of the custom code properties as needed.

Custom Code Property	Description
`CustomInclude`	Specifies a list of folders that contain custom header, source, object, or library files. **Note** If your folder path name contains spaces, you must enclose it in double quotes: `cc.CustomInclude = '"C:\Program Files\MATLAB\work"'`
`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 6-30.

```
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 `coder.ceval`.

For example, from MATLAB code:

```
...
y = 2.5;
y = coder.ceval('myFunc',y);
...
```

See Also

- “Calling Generated C/C++ Functions” on page 6-51

Speeding Up Compilation

In this section...
"Generate Code Only" on page 6-62
"Disable Compiler Optimization" on page 6-62

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` or `C/C++ Executable`.

2 On the **Build** tab, click the `More settings` link.

3 In the Project Settings dialog box **General** tab, set **Compiler optimization level** to `Off`.

At the Command Line

1 Create a code generation configuration object for C/C++ static library or executable. For example, for a library:

```
cfg = coder.config('lib');
```

2 Set the `CCompilerOptimization` to `Off`.

```
cfg.CCompilerOptimization='Off';
```

Code Optimization

In this section...

Unrolling 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. By unrolling short loops with known bounds at compile time, MATLAB generates highly optimized code with no branches.

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 */
}
```

Inlining 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 there are 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;
```

Eliminating 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 improves run-time performance, 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
```

Rewriting 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

MATLAB Coder resolves MATLAB functions by searching first on the code generation path and then on the MATLAB path. By default, unless 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 "How MATLAB Resolves Function Calls in Generated Code" in the Code Generation from MATLAB documentation.

Adding Files to the Code Generation Path

With MATLAB Coder, you can prepend folders and files to the code generation path, as described in "Adding Folders to Search Paths" on page 6-71. By default, this path contains the current folder and the code generation libraries.

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. Because MATLAB Coder searches the code generation path first, a MATLAB file on the code generation path always shadows a MATLAB file of the same name on the MATLAB path. To override a MATLAB function with a with a customized version:

1 Create each version of the MATLAB function in identically–named files.

2 Add the MATLAB version of the function to the MATLAB path.

3 Add the customized version of the function to the code generation path.

   See "Adding Folders to Search Paths" on page 6-71.

Adding Folders to Search Paths

The following table describes how to add folders to search paths. MathWorks recommends that the path 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 folders to:	Do this:
Code generation path using the MATLAB Coder interface	On the MATLAB Coder project **Build** tab: **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.
Code generation path at the command line	Prepend folders to the code generation path using the compiler option `-I`. See `codegen`.
MATLAB path	Follow the instructions in "Adding a Folder to the Search Path" in the MATLAB Programming Fundamentals documentation.

Naming Conventions

MATLAB Coder enforces naming conventions for MATLAB functions and generated files.

- "Reserved Prefixes" on page 6-72
- "Conventions for Naming Generated files" on page 6-72

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 all 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`.

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`

Code Generation for More Than One Entry-Point MATLAB Function

In this section...

Advantages of Generating Code for More Than One Entry-Point Function

Generating a single C/C++ library for more than one entry-point 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, select the input parameter `u`. To the right of this parameter, click the **Actions** icon (⚙) to open the context menu.

ii From the menu, select `Define Type`.

iii In the **Define Type** dialog box, set **Class** to `single`. Click **OK** to specify that the input is a single scalar.

Note MATLAB Coder displays scalars with a size `1x1`.

---

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`.

iv On the **Overview** tab, select the input parameter `u`. To the right of this parameter, click the **Actions** icon (⚙) to open the context menu.

v From the menu, select `Define Type`.

vi In the **Define Type** dialog box, set **Class** to `double`. Click **OK** to specify that the input is a double scalar.

f Repeat step e 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 6-71.

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 6-78.

Generating a C Library with Two Entry-Point Functions Using the Project Interface

You can generate a C library with two entry-point functions, `ep1` and `ep2`, following the same project set-up 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 6-74.) 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 6-78.

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 all 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`.

Example: 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`.

Example: Generating a C/C++ 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 6-174.

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)
```

Example: 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 6-58.
- Generates the C/C++ executable using `codegen`.

See the example, “Example: Calling a C Library Function from C Code” on page 6-55.

Code Generation for Global Data

In this section...

Code Generation 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 "Defining Global Data" on page 6-81.
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 6-82.

Declaring 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;
```

Defining 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 "Declaring Global Variables" on page 6-80 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**.

2. In the Rename Global dialog box **New Name** field, enter the name of the global variable and then click **OK**.

   By default, MATLAB Coder names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, etc.

3 After adding a global variable, before building the project, specify its type and, optionally, initial value. For more information, see “Specifying Global Variable Type and Initial Value in a Project” on page 3-27.

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 “Declaring Global Variables” on page 6-80, 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 6-83.

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 all extrinsic calls. This behavior ensures 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 6-84.

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 `coder.extrinsic -sync:off` option to turn off synchronization for the extrinsic calls that do not affect 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 `coder.extrinsic -sync:on` 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 6-83.

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.

  The **Project Settings** dialog box opens.

2 On the **General** 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 the Global Data Synchronization Mode in the Automatic C MEX Generation Dialog Box.

1 In the MATLAB workspace, define the code generation configuration object. At the MATLAB command line, enter:

```
mexcfg = coder.config('mex');
```

2 Open the Automatic C MEX Generation dialog box:

```
open mexcfg
```

3 On the dialog box **General** tab, set **Global data synchronization mode** as applicable.

4 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 myFile
```

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 affect global data, turn off synchronization for these calls using the `-sync:off` option. For example, if functions `foo1` and `foo2` do not affect 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 affect global data, turn on synchronization for these calls using the `-sync:on` option. For example, if functions `foo1` and `foo2` do affect 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 control the synchronization for specific extrinsic calls. The `-sync:on` option has no effect.

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

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 6-91.

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".

Generating 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` or `C/C++ Executable`, verify that these settings are correct. For more information, see “Changing Output Type” on page 3-35.

3 In the Project Settings dialog box, click the **Comments** tab.

4 On the **Comments** 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
```

Alternatively, you can set this parameter using the MATLAB Coder Project Settings dialog box. On the `Comments` pane , select **MATLAB source code as comments**. For more information, see "Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes" on page 6-28.

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 library:

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('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" in the Embedded Coder documentation.

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" in the Embedded Coder documentation.

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" in the Embedded Coder documentation.

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 any `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 any user comment that precedes 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 any user comment that precedes 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 any user comment that precedes 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 any user comment that precedes 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 any user comment that precedes 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 always includes legally required comments from the MATLAB source code.

Code Generation 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. For information about defining enumerated types based on `int32`, see "Workflows for Using Enumerated Data for Code Generation" in the Code Generation from MATLAB documentation.

Code Generation for Variable-Size Data

In this section...

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. For more information, see "Code Generation for Variable-Size Data". By default, for MEX and C/C++ code generation, support for variable-size data is enabled and dynamic memory allocation is disabled.

Enabling and Disabling Support for Variable-Size Data

By default, for MEX and C/C++ code generation, support for variable-size data is enabled. You can enable dynamic memory allocation 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 **General** tab, select or clear **Enable variable-sizing**. Close the dialog box.

At the Command Line

For MEX Targets. Use the MEX configuration object `EnableVariableSizing` option. For example, to disable variable sizing:

1 Create a MEX configuration object and set the `EnableVariableSizing` option:

```
mexcfg = coder.config('mex');
mexcfg.EnableVariableSizing = false;
```

2 Using the `-config` option, pass the configuration object to `codegen`. For example, to disable variable sizing and generate a MEX function for a function, `foo`, that takes no inputs:

```
codegen -config mexcfg foo
```

For C/C++ Executable Targets. Use the code generation configuration object `EnableVariableSizing` option. For example, to disable variable sizing and generate an executable for a function, `foo`, that takes no inputs:

1 Create a code generation configuration object and set the `EnableVariableSizing` option:

```
cfg = coder.config('exe');
cfg.EnableVariableSizing = false;
```

2 Using the `-config` option, pass the configuration object to `codegen` :

```
codegen -config cfg foo
```

Enabling and Disabling Dynamic Memory Allocation

By default, dynamic memory allocation is disabled. To use dynamic memory allocation, enable support for variable-size data (see "Enabling and Disabling Support for Variable-Size Data" on page 6-99). You can enable dynamic memory allocation 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 **General** tab, set **Dynamic memory allocation** to `For all variable-sized arrays` or `Off`. Close the dialog box.

At the Command Line

For MEX Targets. Use the MEX configuration object `DynamicMemoryAllocation` option. For example, to enable dynamic memory allocation and generate a MEX function for a function, `foo`, that takes no inputs:

1. Create a MEX configuration object and set the `DynamicMemoryAllocation` option:

   ```
   mexcfg = coder.config('mex');
   mexcfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
   ```

2. Using the `-config` option, pass the configuration object to `codegen`:

   ```
   codegen -config mexcfg foo
   ```

For C/C++ Executable Targets. Use the code generation configuration object `DynamicMemoryAllocation` option. For example, to enable dynamic memory allocation and generate an executable for a function, `foo`, that takes no inputs:

1. Create a code generation configuration object and set the `DynamicMemoryAllocation` option:

```
cfg = coder.config('exe');
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

2 Using the `-config` option, pass the configuration object to `codegen` :

```
codegen -config cfg foo
```

Generating Code for MATLAB Functions with Variable-Size Data

Here is a basic workflow that recommends first generating MEX code for verifying the generated code and then generating standalone code after you are satisfied with the result of the prototype.

To work through these steps with a simple example, see "Generating Code for a MATLAB Function That Expands a Vector in a Loop" on page 6-104

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” in the Code Generation from MATLAB documentation.

5 Fix upper bounds errors

Cause:	How To Fix:	For More Information:
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	In the Code Generation from MATLAB documentation, see "Specifying Upper Bounds for Variable-Size Data" and "Diagnosing and Fixing Size Mismatch Errors".
MATLAB attempts to compute an upper bound for unbounded variable-size data.	If the data is unbounded, enable dynamic memory allocation.	See "Enabling and Disabling Dynamic Memory Allocation" on page 6-100.

6 Generate C/C++ code using the `codegen` function.

Generating Code for a MATLAB Function That Expands a Vector in a Loop

About the MATLAB Function uniquetol

This tutorial 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`, `abs(B(i) - B(j))` > `tol` for all `i` and `j`. Initially, assume input vector `A` can store up to 100 elements. In a later exercise, you will enable dynamic memory allocation for an unbounded input vector.

```
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
```

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),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 "Specifying Variable-Size Inputs at the Command Line" on page 6-38 in the MATLAB Coder documentation.
   - The second argument, `0`, defines input `tol` as a real double scalar.

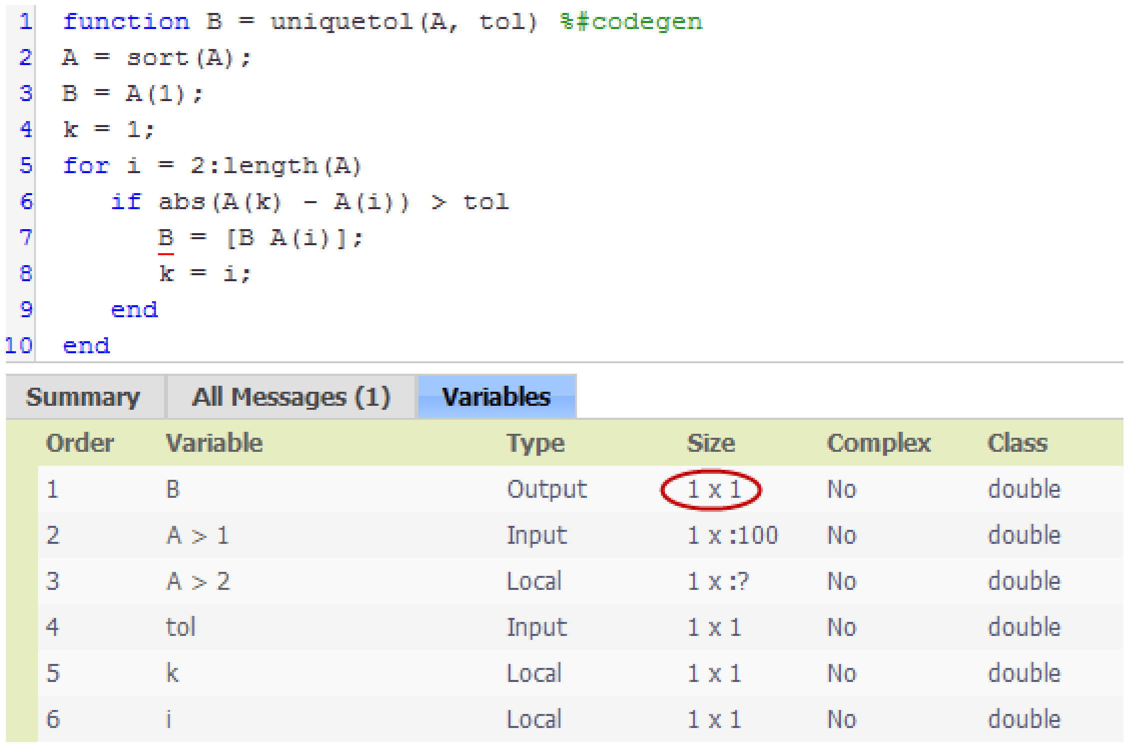
   The `-report` option instructs `codegen` to generate a code generation report, even if no errors or warnings occur.

   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.



```
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
```

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.

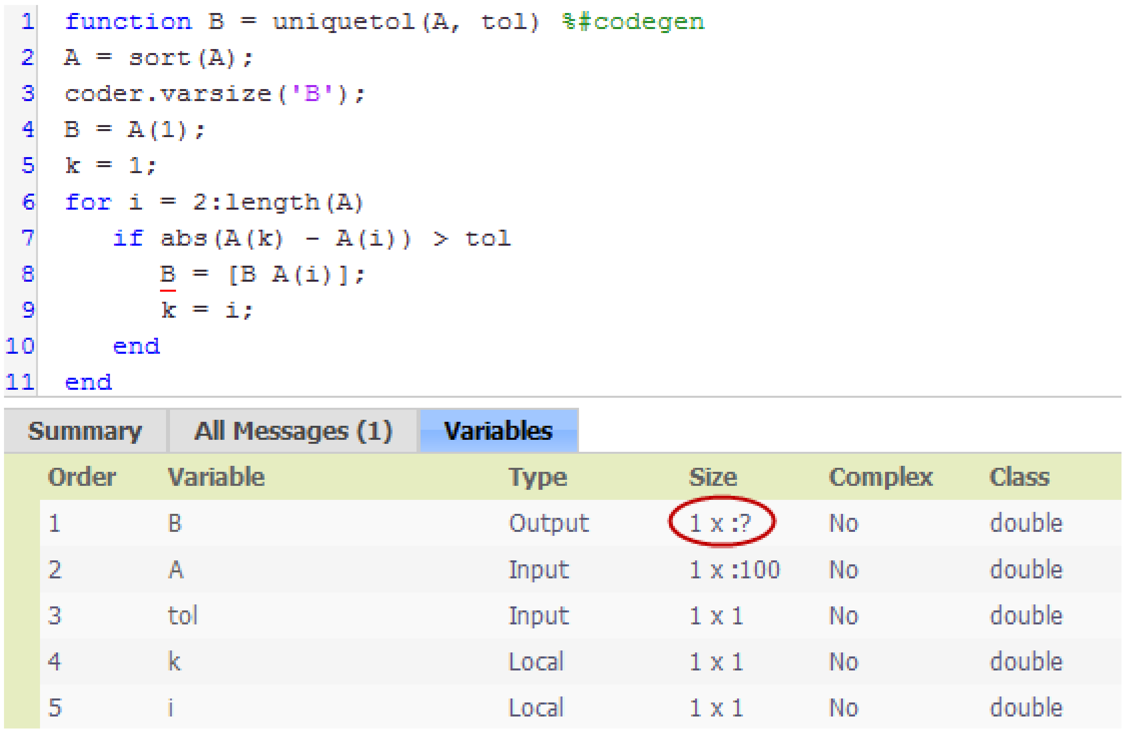
2 Generate another compilation report using this command:

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

This time, the command generates a different compiler error:

```
??? 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. Please consider enabling
dynamic memory allocation to allow unbounded sizes.
```

3 Open the error report and select the Variables tab.



```
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
```

Order	Variable	Type	Size	Complex	Class
1	B	Output	1 x :?	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

This error occurs because `codegen` cannot determine an upper bound for `B` .

Step 5: Fix the Upper Bounds Error

There are two ways to fix this error:

- "Specify Upper Bounds for Variable `B`" on page 6-109
- "Enable Dynamic Memory Allocation for an Unbounded Variable `B`" on page 6-110

Specify Upper Bounds for Variable `B`. Choose this method if you want to enforce an upper bound for `B`. In this exercise, you constrain `B` to the same upper bound as `A`.

1 Add a second argument to `coder.varsize`:

```
coder.varsize('B', [1 100]);
```

The argument `[1 100]` specifies that `B` is a vector with its first dimension fixed at size 1 and the second dimension variable to an upper bound of 100. The value of 100 matches the upper bound of variable-size vector `A`. Based on the algorithm, output `B` is at most as large as input `A`. By default, dimensions of 1 are fixed size.

Here is the modified code:

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);

coder.varsize('B', [1 100]);

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
```

2 Compile the function again:

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

`codegen` should compile successfully and, in the current folder, generate a MEX function, `uniquetol_mex`, for `uniquetol`.

This exercise presents one way to specify an upper bound. To learn about other methods, see “Specifying Upper Bounds for Variable-Size Data” in the Code Generation from MATLAB documentation.

Enable Dynamic Memory Allocation for an Unbounded Variable `B`. Choose this method if you do not know the upper bound for `B` or do not need to enforce an upper bound. In this exercise, you will also remove the upper bound for input `A`.

1 Enable the `DynamicMemoryAllocation` configuration option for MEX code generation:

```
cfg = coder.config('mex');
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

2 Compile the function again, adding a `-config` argument:

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

Adding `-config cfg` applies the configuration setting that enables dynamic memory allocation.

`codegen` should compile successfully and generate a MEX function for `uniquetol`.

3 Specify the second dimension of input A as unbounded:

```
codegen -config cfg -report uniquetol -args
{coder.typeof(0,[1 Inf]), 0}
```

If you do not know the upper bounds of an input, it is good coding practice to specify the input as unbounded instead of giving it an arbitrary upper bound. In this `codegen` command, the size of the second dimension of input `A` is `Inf`. When you specify the size of a dimension as `Inf` in a `coder.typeof` statement, `codegen` treats the dimension as unbounded. You can use `Inf` only with dynamic allocation.

See Also.

- `codegen`
- "Primary Function Input Specification" on page 6-30

Step 6: Generate C/C++ Code

Generate C/C++ code for variable-size data in the example function. For example, to generate a C library for unbounded data in the `emldemo_uniquetol` function using dynamic memory allocation:

1. Enable the `DynamicMemoryAllocation` configuration option for C library generation:

   ```
   cfg=coder.config('lib');
   cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
   ```

2. Issue this command:

   ```
   codegen -config cfg -report uniquetol -args {coder.typeof(0,[1 Inf]),0}
   ```

In the generated code, MATLAB represents data with unknown upper bounds as a data type called `emxArray`. MATLAB provides utility functions for creating and interacting with `emxArrays` in your generated code.

See Also.

- "C Code Interface for Unbounded Arrays and Structure Fields" in the Code Generation from MATLAB documentation.

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
```

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)}
```

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
```

```
.                          run_atoms.obj
..                         run_atoms_emxAPI.c
buildInfo.mat              run_atoms_emxAPI.h
rtGetInf.c                 run_atoms_emxAPI.obj
rtGetInf.h                 run_atoms_emxutil.c
rtGetInf.obj               run_atoms_emxutil.h
rtGetNaN.c                 run_atoms_emxutil.obj
rtGetNaN.h                 run_atoms_initialize.c
rtGetNaN.obj               run_atoms_initialize.h
rt_nonfinite.c             run_atoms_initialize.obj
rt_nonfinite.h             run_atoms_ref.rsp
rt_nonfinite.obj           run_atoms_rtw.bat
rtw_proj.tmw               run_atoms_rtw.mk
rtwtypes.h                 run_atoms_terminate.c
run_atoms.c                run_atoms_terminate.h
run_atoms.h                run_atoms_terminate.obj
run_atoms.lib              run_atoms_types.h
run_atoms.lnk
```

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"

void 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-
        }

        /* Free memory */
        emxDestroyArray_Atom(atoms);
    }
```

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]),in
```

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

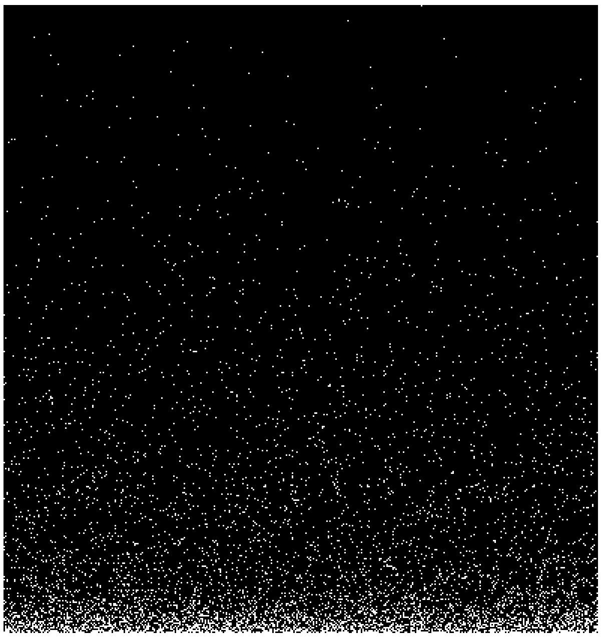
```
load atoms_state.mat -ascii
clear atoms
```

```
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
```

How MATLAB Coder Partitions Generated Code

In this section…

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 6-129).

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 6-120. 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 **General** tab.

4 On the **General** 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

For MEX Targets. Use the MEX 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
```

For C/C++ Executable Targets. Use the `coder.CodeConfig` configuration object `FilePartitionMethod` option. For example, to compile the function`foo` that has no inputs and generate all C/C++ functions into a single file:

1 Create a code generation configuration object and set the `FilePartitionMethod` option:

```
cfg = coder.config('exe');
cfg.FilePartitionMethod = 'SingleFile';
```

2 Using the `-config` option, pass the configuration object to `codegen`.

```
codegen -config cfg foo
```

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 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 3-7.

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 **Advanced** tab.

   d Set the **Inline threshold** parameter to `0`.

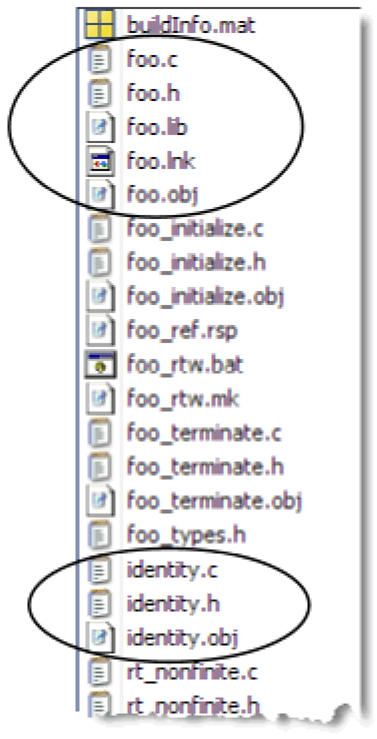
4 Click **Build** to generate a library.

To generate a C 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.



```
buildInfo.mat
foo.c
foo.h
foo.lib
foo.lnk
foo.obj
foo_initialize.c
foo_initialize.h
foo_initialize.obj
foo_ref.rsp
foo_rtw.bat
foo_rtw.mk
foo_terminate.c
foo_terminate.h
foo_terminate.obj
foo_types.h
identity.c
identity.h
identity.obj
rt_nonfinite.c
rt_nonfinite.h
```

How MATLAB Coder Partitions Subfunctions

For each subfunction, 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 subfunction `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, make sure you select a C++ compiler and select C++ as your target language. For example, at the command line:

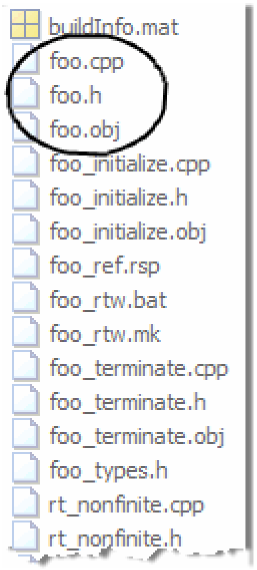
1 Select C++ as your target language:

```
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` subfunction.



buildInfo.mat
foo.cpp
foo.h
foo.obj
foo_initialize.cpp
foo_initialize.h
foo_initialize.obj
foo_ref.rsp
foo_rtw.bat
foo_rtw.mk
foo_terminate.cpp
foo_terminate.h
foo_terminate.obj
foo_types.h
rt_nonfinite.cpp
rt_nonfinite.h

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`:

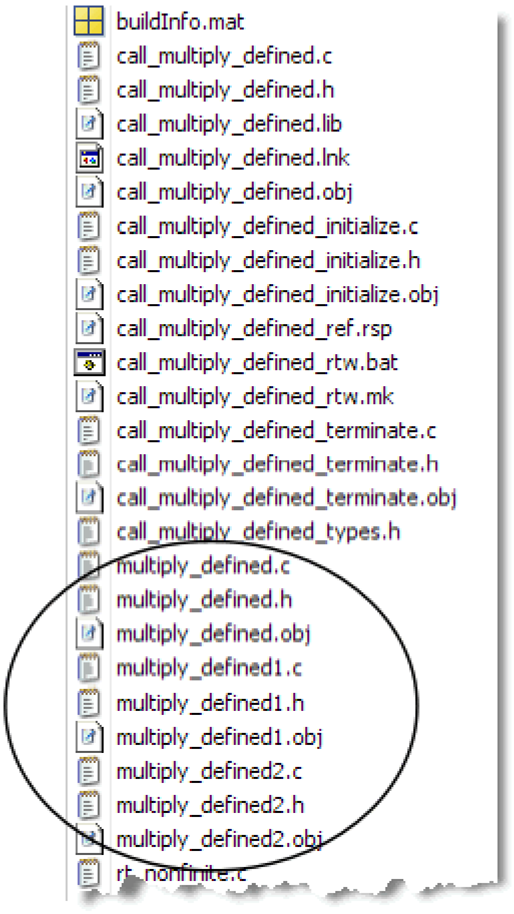
```
%#codegen
function [y1,y2,y3] = call_multiply_defined

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.



```
buildInfo.mat
call_multiply_defined.c
call_multiply_defined.h
call_multiply_defined.lib
call_multiply_defined.lnk
call_multiply_defined.obj
call_multiply_defined_initialize.c
call_multiply_defined_initialize.h
call_multiply_defined_initialize.obj
call_multiply_defined_ref.rsp
call_multiply_defined_rtw.bat
call_multiply_defined_rtw.mk
call_multiply_defined_terminate.c
call_multiply_defined_terminate.h
call_multiply_defined_terminate.obj
call_multiply_defined_types.h
multiply_defined.c
multiply_defined.h
multiply_defined.obj
multiply_defined1.c
multiply_defined1.h
multiply_defined1.obj
multiply_defined2.c
multiply_defined2.h
multiply_defined2.obj
rt_nonfinite.c
```

For more information, see “Overloaded MATLAB Functions” in the MATLAB Programming Fundamentals documentation.

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	`codegen/mex/`*`function_name`*
HTML reports	`codegen/mex/`*`function_name`*`/html`

Generated Files for C/C++ Library Targets

By default, MATLAB Coder generates the following files for C/C++ library targets.

Type of Files	Location
C/C++ source, library, header, and object files	`codegen/lib/`*`function_name`*
HTML reports	`codegen/lib/`*`function_name`*`/html`

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/`*`function_name`*
HTML reports	`codegen/exe/`*`function_name`*`/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	On the **Build** tab: **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...
	Note MathWorks recommends that the output file location 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.

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 any 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 6-121.
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 does not inline all functions, only those 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 6-120.
- 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 **Advanced** tab.

4 On this tab, 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);
```

The 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 6-132.

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 **Advanced** pane, set the value of the **Inline threshold** parameter.

Modifying the Inlining Threshold at the Command Line. Set the value of the `InlineThreshold` parameter for the appropriate configuration object. See `coder.MexCodeConfig`, `coder.CodeConfig`, `coder.EmbeddedCodeConfig`.

Customizing the Post-Code-Generation Build Process

In this section...

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 6-135 to access this information in the build information object. “Programming a Post-Code-Generation Command” on page 6-170 explains how to use the functions to control a post-code-generation build.

Build Information Functions

- "getNonBuildFiles" on page 6-164
- "getSourceFiles" on page 6-166
- "getSourcePaths" on page 6-168
- "updateFilePathsAndExtensions" on page 6-169
- "updateFileSeparator" on page 6-170

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 *`groups`* as a...
Apply one group name to all 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 *`options`*.

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 all 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 *macrodefs*.

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 all 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 `filenames`.

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 all 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 *`filenames`*.

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 *`groups`* as a...
Apply one group name to all 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 *`paths`*.

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 all 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 *paths*.

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 all 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 *options*.

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)
```

All arguments except *buildinfo* , *linkobjs*, and *paths* are optional. If you specify an optional argument, you must specify all of 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 *`groups`* as a...
Apply one group name to all 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 *`linkobjs`*.

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 *`paths`* or *`groups`* as a...	The Function...
Character array	Applies the character array to all 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 *`linkobjs`*.

Similarly, you can specify *`priority`*, *`precompiled`*, and *`linkonly`* as a value or vector of values.

If You Specify *priority*, *precompiled*, or *linkonly* as a...	The Function...
Value	Applies the value to all 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 *linkobjs*.

If you choose to specify an optional argument, you must specify all of the optional arguments preceding it. For example, to specify that all 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 all 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 `filenames`.

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 all 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 *`filenames`*.

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 *group* as a...
Apply one group name to all 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 *filenames*.

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 all 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 *filenames*.

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 *`groups`* as a...
Apply one group name to all 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 *`paths`*.

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 all 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 *`paths`*.

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 all 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 *tokennames*.

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 take appropriate action with 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 all 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 *tokennames*.

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 any 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 all 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
macrodefs	Complete macro definitions with `-D` prefix
identifiers	Names of the macros
values	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 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 not necessary 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...
`true`	Concatenates and returns each filename with its corresponding path.
`false`	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...
`true`	Concatenates and returns each filename with its corresponding path.
`false`	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 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...
`true`	Concatenates and returns each filename with its corresponding path.
`false`	Returns only filenames.

Note Usually it is not necessary 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...
`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 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'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 source file paths the function returns.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null string (`''`) for *`includeGroups`*.

updateFilePathsAndExtensions

Purpose. Update files in project's 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` would be updated to `myfile.c`.

Description. Using paths that already exist in a project's build information, the `updateFilePathsAndExtensions` function checks whether any 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 all file path specifications.

Description. The `updateFileSeparator` function changes all 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 any combination of build information functions to customize the post-code-generation build. See "Example: Programming and Using a Post-Code-Generation Command at the Command Line" on page 6-173

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 6-172 and "Including a Post-Code-Generation Command in the Project Settings Dialog Box." on page 6-171.

Including a Post-Code-Generation Command at the Command Line

Set the `PostCodeGenCommand` option for the appropriate 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 6-172 and "Including a Post-Code-Generation Command in the Project Settings Dialog Box." on page 6-171.

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)';
```

Example: 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

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.

Even when there are no errors or warnings, you can also use the option to request reports.

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 "Specifying Output File Locations" on page 3-34.

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. For successful builds, 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 all errors and warnings.

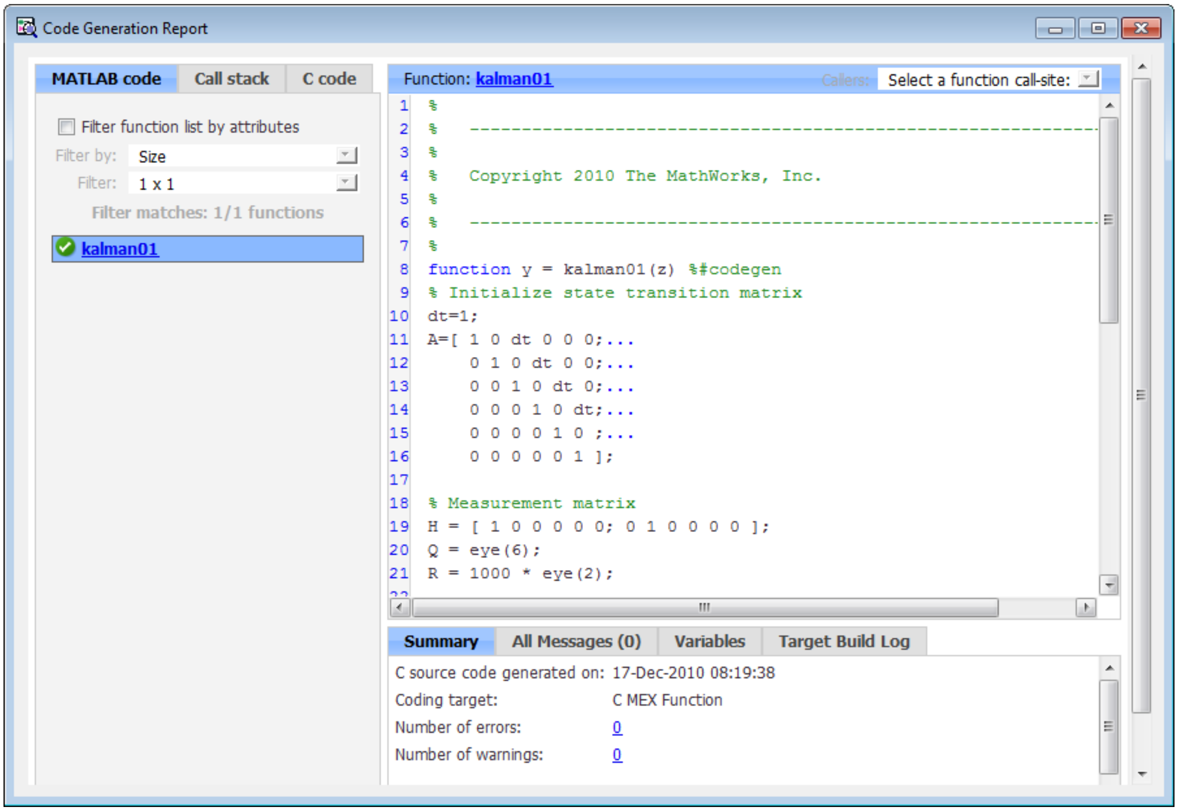
Opening Code Generation Reports at the Command Line. If you specify the `-launchreport` option, the code generation report opens automatically.

If no build errors occurred, To open the code generation report, in the MATLAB Command Window, click the **`View report`** link.

If build errors occurred, 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 successful build.



```
%
%   ----------------------------------------------------------------
%
%   Copyright 2010 The MathWorks, Inc.
%
%   ----------------------------------------------------------------
%
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 ];

% Measurement matrix
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
Q = eye(6);
R = 1000 * eye(2);
```

```
C source code generated on: 17-Dec-2010 08:19:38
Coding target:              C MEX Function
Number of errors:           0
Number of warnings:         0
```

The report provides the following information, as applicable:

- MATLAB code information, including a list of all functions 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 all error and warning messages
- List of all variables in your MATLAB code
- Target build log that records compilation and linking activities

How to 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 **Report** tab.
3. On the **Report** 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++ 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.

Viewing 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 highlighted in the list on this tab.

The **MATLAB code** tab provides:

- A list of the MATLAB functions that have been built. To indicate whether the build was successful, the report displays icons next to each function name:
  - ❌ Errors in function.
  - ⚠ Warnings in function.
  - ✅ Successful build, no errors or warnings.
- A filter control. You can use **Filter function list by attributes** to sort your functions by:
  - Size
  - Complexity
  - Class

Viewing Subfunctions

The code generation report annotates the subfunction 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 subfunction `subfcn`, and `fcn2` contains the subfunction `subfcn2`, the report displays:

```
fcn1 > subfcn1
fcn2 > subfcn2
```

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 Call Stack Information

The code generation report provides call stack information:

- On the **Call stack** tab.
- In the list of **Callers**.

  If a function is called from more than one function, this list provides details of each call-site. Otherwise, the list is disabled.

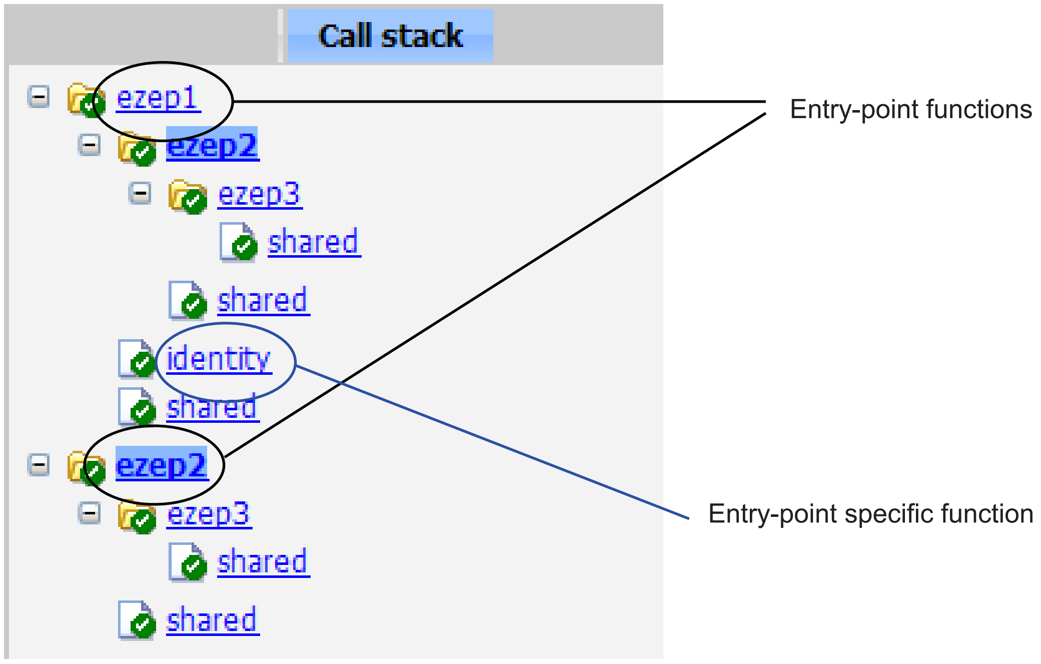
Viewing Call Stack Information on the Call stack Tab

To view call stack information, click the **Call stack** tab.

The call stack lists the functions in the order that the top-level function calls them. It also lists the subfunctions that each function calls.

If there is 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 all instances of this function. If you click an entry-point specific function, the report highlights only that instance.

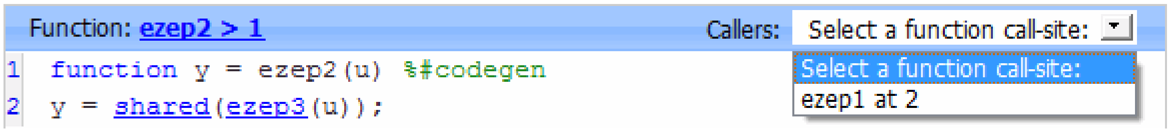
For example, in the following call stack, `ezep1` and `ezep2` are entry-point functions. `identity` is an entry-point specific function, called only by `ezep1`. Functions `ezep3` and `shared` are shared functions.



Viewing Function Call-Sites in the Callers List

If a function is called from more than one function, the **Callers** list provides details of each call site. To navigate between call-sites, select a call-site from the **Callers** 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 **Callers** list displays the other call-site in `ezep1`.



```
function y = ezep2(u) %#codegen
y = shared(ezep3(u));
```

Viewing 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 6-88.

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 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.

Viewing 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 all errors and warnings on the **All Messages** tab. The report lists these messages in chronological order.
- Highlighting all 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 all 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 all 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" in the Fixed-Point Toolbox documentation.
- 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 "Working with Fixed-Point Code Generation Reports" in the Fixed-Point Toolbox documentation.

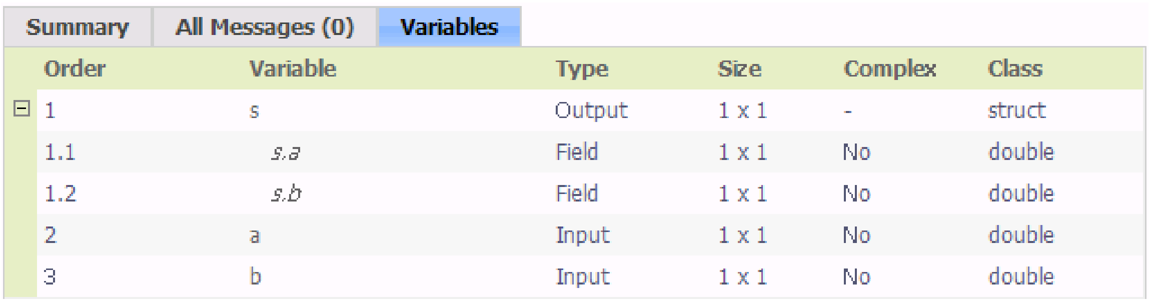
It only displays a column if at least one variable in the code has information in that column. For example, if the code does not contain any fixed-point data types, the report does not display the `DT mode`, `WL` or `FL` columns.

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.



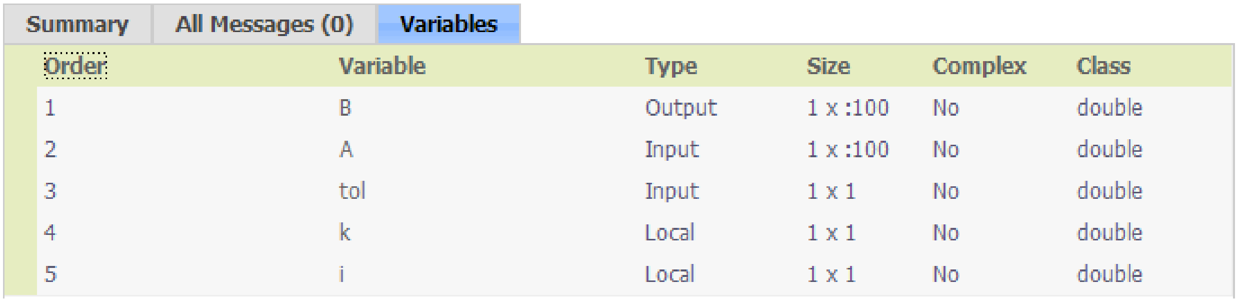
Summary | All Messages (0) | Variables

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 **:**.

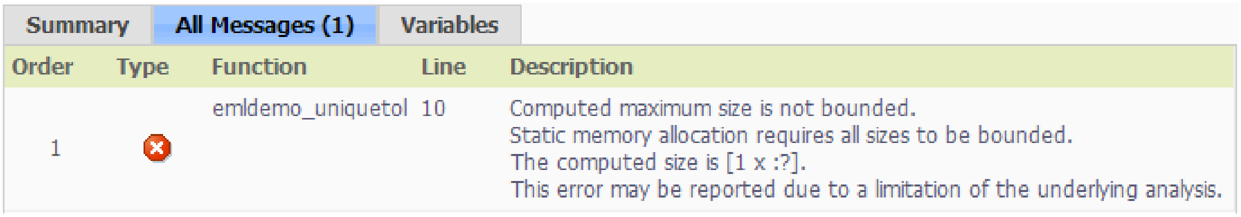
In the following report, variable *A* is variable-size. Its maximum computed size is 1×100.



Summary | All Messages (0) | Variables

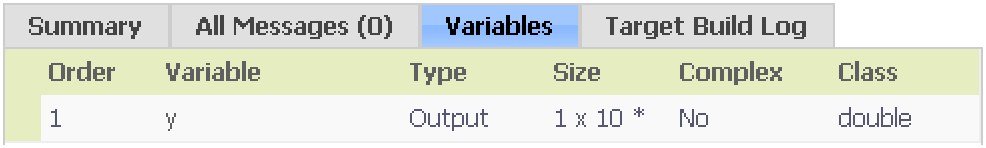
Order	Variable	Type	Size	Complex	Class
1	B	Output	1 x :100	No	double
2	A	Input	1 x :100	No	double
3	tol	Input	1 x 1	No	double
4	k	Local	1 x 1	No	double
5	i	Local	1 x 1	No	double

If the code generation software cannot compute the maximum size of a variable-size array, the report displays the size as **:?**.



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.

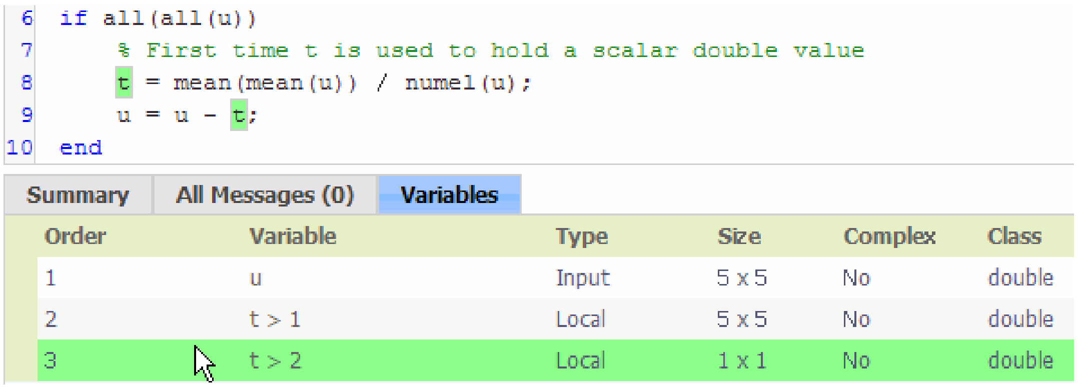


Order	Variable	Type	Size	Complex	Class
1	y	Output	1 x 10 *	No	double

For more information on how to use the size information for variable-sized arrays, see “How Working with Variable-Size Data Is Different for Code Generation” in the Code Generation from MATLAB documentation.

Viewing Renamed Variables in the Variables Tab. If your MATLAB function reuses a variable with different size, type, or complexity, whenever possible, the code generation software creates separate, uniquely named variables in the generated code. For more information, see “Reusing the Same Variable with Different Properties” in the Code Generation from MATLAB documentation. 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.



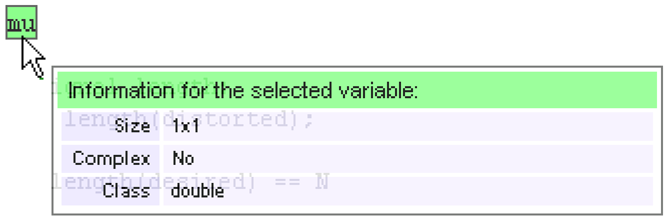
```
if all(all(u))
    % First time t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
```

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:

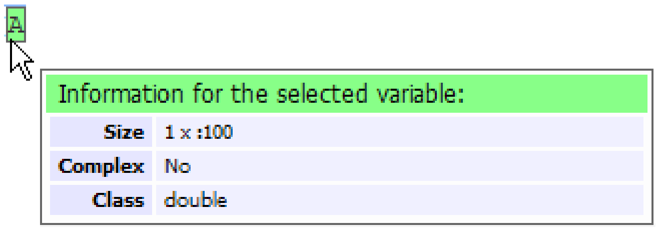
Green, when the variable has data type information at this location in the code.



Information for the selected variable:

Size	1x1
Complex	No
Class	double

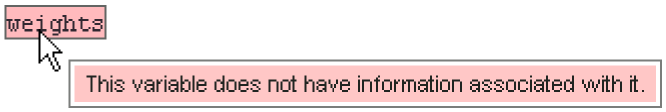
For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon **:**. Here the array `A` is variable-sized with a maximum computed size of `1 x 100`.



Information for the selected variable:

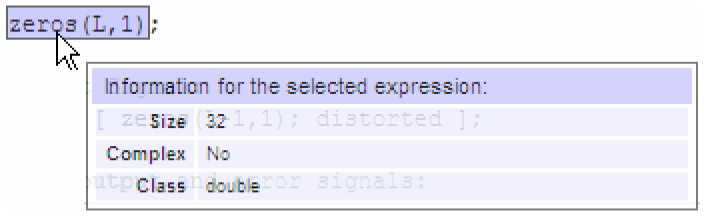
Size	1 x :100
Complex	No
Class	double

Pink, when the variable has no data type information.



This variable does not have information associated with it.

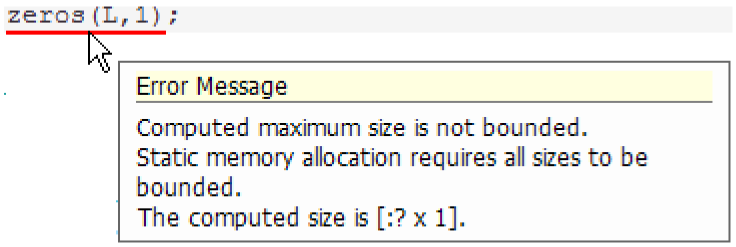
Purple, information about expressions. You can also view information about expressions in your MATLAB code. On the MATLAB code pane, place your pointer over an expression . The report highlights expressions in purple and provides more detailed information.



Information for the selected expression:

Size	32
Complex	No
Class	double

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.



Error Message

Computed maximum size is not bounded.
Static memory allocation requires all sizes to be bounded.
The computed size is [:? x 1].

Viewing Target Build Information

If MATLAB Coder builds your MATLAB code successfully, it provides target build information on the **Target Build Log** tab, including:

- Build folder

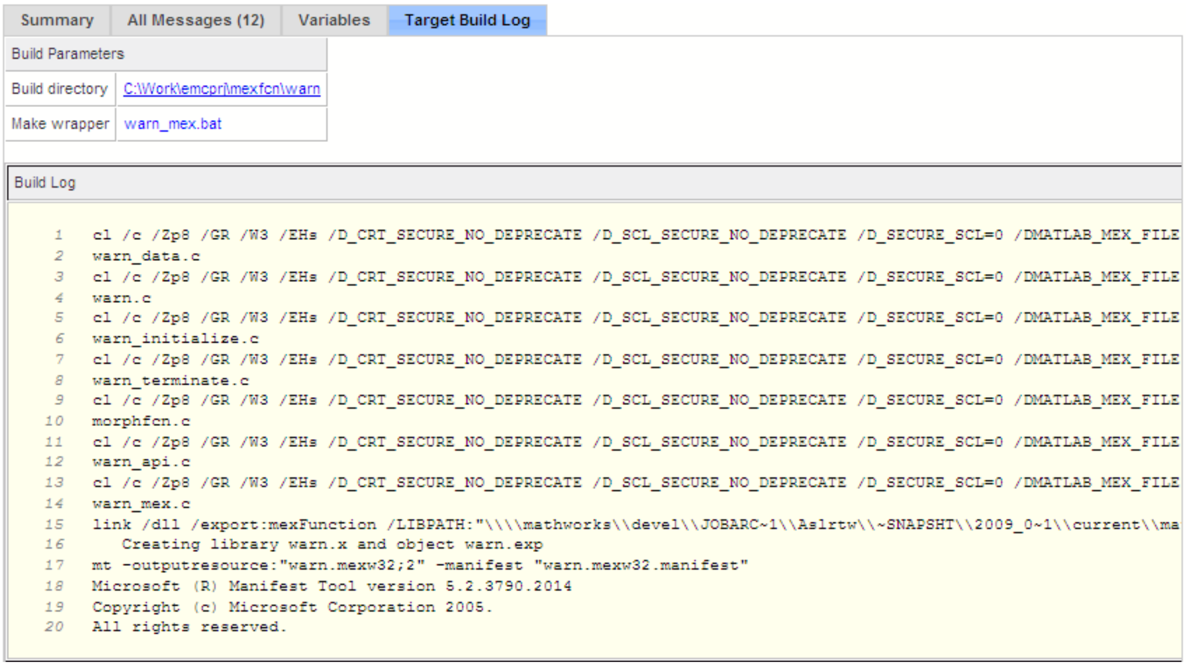
  Clicking this link changes the MATLAB current folder to the build folder.

- 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.



Build Parameters	
Build directory	C:\Work\emcprj\mexfcn\warn
Make wrapper	warn_mex.bat

Build Log

```
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
warn_data.c
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
warn.c
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
warn_initialize.c
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
warn_terminate.c
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
morphfcn.c
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
warn_api.c
cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
warn_mex.c
link /dll /export:mexFunction /LIBPATH:"\\\\mathworks\\devel\\JOBARC~1\\Aslrtw\\~SNAPSHT\\2009_0~1\\current\\ma
   Creating library warn.x and object warn.exp
mt -outputresource:"warn.mexw32;2" -manifest "warn.mexw32.manifest"
Microsoft (R) Manifest Tool version 5.2.3790.2014
Copyright (c) Microsoft Corporation 2005.
All rights reserved.
```

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 the correct information for unrolled loops.

Dead Code

The report does not display information about any 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.

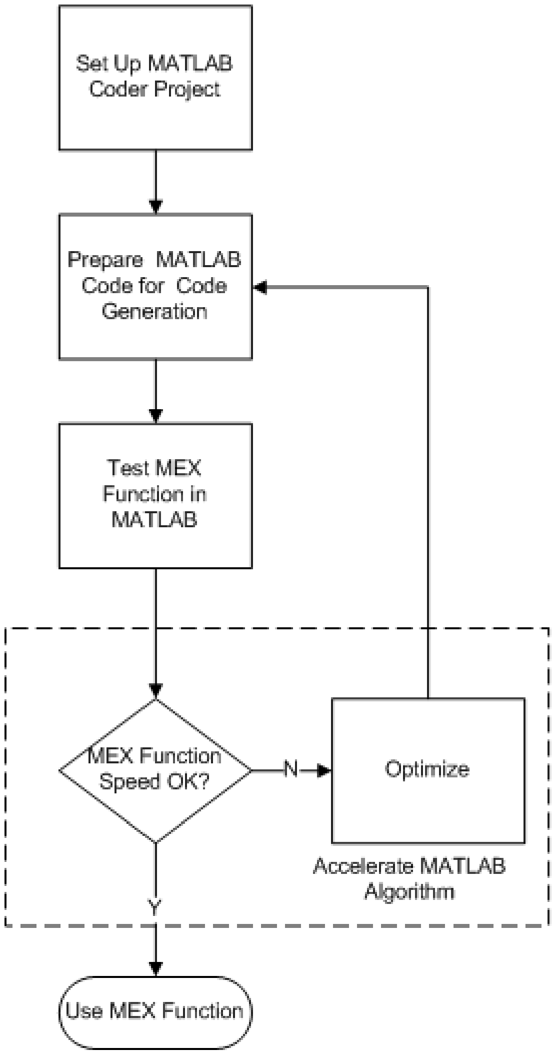
```
out1 = [1 2 3;
        4 5 6];
```

The report does support selection of single line matrices.

```
out1 = [1 2 3; 4 5 6];
```

Accelerating MATLAB Algorithms

Workflow for Accelerating MATLAB Algorithms


Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Test MEX Function in MATLAB
MEX Function Speed OK?
N
Optimize
Accelerate MATLAB Algorithm
Y
Use MEX Function

See Also

- Chapter 3, “Setting Up a MATLAB® Coder Project”
- Chapter 4, “Preparing MATLAB Code for C/C++ Code Generation”
- Chapter 5, “Testing MEX Functions in MATLAB”
- “Modifying MATLAB Code for Acceleration” on page 7-5
- “Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms” in the Fixed-Point Toolbox documentation

How to Accelerate MATLAB Algorithms

For many applications, you can generate MEX functions to accelerate MATLAB algorithms. If you have a Fixed-Point Toolbox 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 7-5
- "Speeding Up MATLAB Algorithms with the Basic Linear Algebra Subprograms (BLAS) Library" on page 7-11
- "Controlling Run-Time Checks" on page 7-14

Modifying MATLAB Code for Acceleration

In this section...

How to Modify Your MATLAB Code for Acceleration

You might improve the efficiency of the generated code using one of the following optimizations:

- "Unrolling for-loops" on page 6-64
- "Inlining Code" on page 6-66
- "Eliminating Redundant Copies of Function Inputs (A=foo(A))" on page 6-67

Unrolling 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. By unrolling short loops with known bounds at compile time, MATLAB generates highly optimized code with no branches.

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 */
  }
```

Inlining 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 there are 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;
```

Eliminating 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 improves run-time performance, 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
```

Speeding Up MATLAB Algorithms with the Basic Linear Algebra Subprograms (BLAS) Library

In this section...

How MATLAB Uses the BLAS Library for MEX Code Generation

The Basic Linear Algebra Subprograms (BLAS) Library is a library of external linear algebra routines optimized for fast computation of low-level matrix operations. By default, MATLAB functions call BLAS library routines to accelerate MEX function execution, except in these cases:

- Your C/C++ compiler does not support the BLAS library.
- The size of the matrix is below a minimum threshold.

  MATLAB for code generation uses a heuristic to evaluate matrix size against the overhead of calling an external library.

How to Use the BLAS Library for C/C++ Code Generation

If you have an Embedded Coder license, you can set up MATLAB Coder to use a Target Function Library to map the following operations to a BLAS Subroutine:

- Matrix multiplication
- Matrix multiplication with transpose on single or both inputs
- Matrix multiplication with Hermitian operation on single or both inputs

To run a demo showing you how to do this, see Replacing Math Functions and Operators.

Note Requires an Embedded Coder license.

See Also

- “Code Replacement” in the Embedded Coder documentation.

When to Disable BLAS Library Support

Consider disabling BLAS library support for MATLAB functions when:

- You are executing code on a 64-bit platform and the number of elements in a matrix exceeds 32 bits.

  MATLAB Coder automatically truncates the matrix size to 32 bits.
- Your platform does not provide a robust implementation of BLAS routines.

How to Disable BLAS Library Support

MATLAB Coder software enables BLAS library support by default. However, you can disable this feature explicitly from the project settings dialog box, the command line, or a MEX configuration dialog box.

Disabling BLAS Library Support in the Project Settings Dialog Box

1. On the MATLAB Coder project **Build** tab, click `More settings`.

   The **Project Settings** dialog box opens.

2. On the **General** tab, clear **Enable BLAS library if possible**.

Disabling BLAS Library Support at the Command Line

1. In the MATLAB workspace, define the MEX configuration object by issuing a constructor command, like this:

```
mexcfg = coder.config('mex');
```

2 Disable the BLAS option.

```
mexcfg.EnableBLAS = false;
```

Disabling BLAS Library Support in the Automatic C MEX Generation Dialog Box

1 In the MATLAB workspace, define the MEX configuration object:

```
mexcfg = coder.config('mex');
```

2 Open the Automatic C MEX Generation dialog box:

```
open mexcfg
```

3 On the dialog box **General** tab, clear **Enable BLAS library if possible** and click **Apply**.

Supported Compilers

MATLAB Coder uses the BLAS library on all C/C++ compilers **except**:

- Watcom
- Intel
- Borland

The default MATLAB compiler for Windows 32–bit platforms, `lcc`, supports the BLAS library, but it is not designed to generate high performance code. To install a different C/C++ compiler, use the `mex -setup` command, as described in “Building MEX-Files” in the MATLAB External Interfaces documentation.

Controlling Run-Time Checks

In this section...

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".

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 have an adverse effect on performance. 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 all 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`.

   The **Project Settings** dialog box opens.

2 On the **General** tab, clear **Ensure memory integrity**, **Ensure responsiveness**, 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;
```

Disabling Run-Time Checks in the Automatic C MEX Generation Dialog Box

1 In the MATLAB workspace, define the MEX configuration object:

```
mexcfg = coder.config('mex');
```

2 Open the Automatic C MEX Generation dialog box:

```
open mexcfg
```

3 On the dialog box **General** tab, clear **Ensure memory integrity**, **Ensure responsiveness**, or **Extrinsic calls** , as applicable, and click **Apply**.

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 demo, 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 The MathWorks, Inc.
% The seed will guarantee that we get the precisely same simulation
% every time we call this function.
rand('seed', 1283);

% The 'cdata' variable is a matrix representing the colordata bitmap whic
% 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
```

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
```

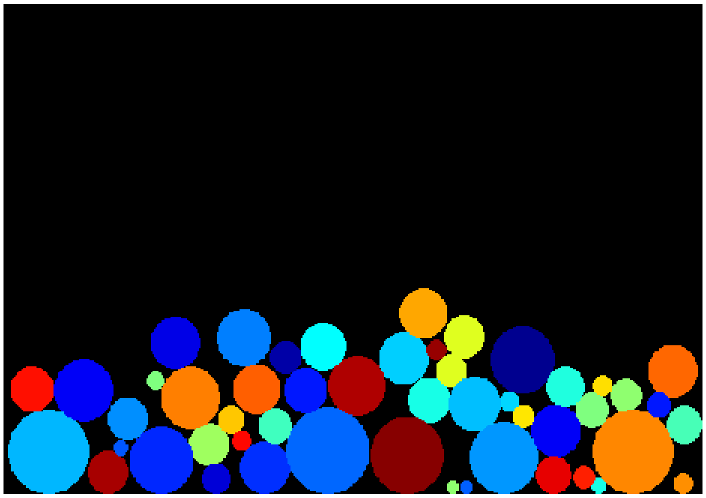
The ’run_balls’ function calls other MATLAB functions, but you need to specify only the entry-point function when calling ’codegen’.

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 ~19.3
```

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

In this section...

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 "Calling External C/C++ Functions" on page 8-3)
- Passing arguments by reference to C/C++ code (see "Passing Arguments by Reference to External C/C++ Functions" on page 8-3)
- Manipulating C/C++ data (see "Declaring Data" on page 8-5)

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.

.

Calling 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.

Passing 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.

  By using `coder.wref`, you can achieve the effect of returning 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)));
```

Declaring 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`.

Example: 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 all 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 all 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;
```

Calling External C/C++ Functions

In this section…
“Workflow for Calling External C/C++ Functions” on page 8-7
“Best Practices for Calling C/C++ Code from Generated Code” on page 8-8

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 necessary.

   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 8-10.

   Tip One way to add these type definitions is to include the header file `tmwtypes.h`, which defines all 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 needed (see “Passing Arguments by Reference to External C/C++ Functions” on page 8-3).

4. Include the custom C/C++ functions in the build. See “Custom C/C++ Code Integration” on page 6-58.

5. Check that there are no compilation warnings about data type mismatches.

This step ensures that you catch type mismatches between C/C++ and MATLAB (see "How MATLAB® Coder Infers C/C++ Data Types" on page 8-10).

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 each call to the C/C++ function has the correct type.
- 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 6-58.

Returning 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.ref(a), coder.wref(b), coder.wref(c));
```

How MATLAB Coder Infers C/C++ Data Types

In this section...

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 *`
`single`	`real32_T`	`real32_T *`

MATLAB Type	C/C++ Type	C/C++ Reference Type
`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 8-11.	
`struct`	Fields also affect the C/C++ type. See "Mapping Structures to C/C++ Structures" on page 8-13 .	
`complex`	See "Mapping embedded.numerictypes to C/C++" on page 8-11.	
function handles	Not supported.	

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 Toolbox 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
`numerictype(1, 16, 15)`	`int16_T`	`int16_T *`
`numerictype(1, 13, 10)`	`int16_T`	`int16_T *`
`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
`zeros(10, 5, 'int8')`	`int8_T *`	`int8_T *`
`zeros(5, 10, 'int8')`	`int8_T *`	`int8_T *`
`zeros(3, 7)`	`real_T *`	`real_T *`
`zeros(10, 1, 'single')`	`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 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.

Examples

Use this list to find examples in the documentation.

Generating MEX Functions

Generating Static C/C++ Libraries

Generating C/C++ Executables

Specifying Inputs

Calling C/C++ Code from MATLAB Code

Optimizing Generated Code

Generating Code for Variable-Size Data

Index