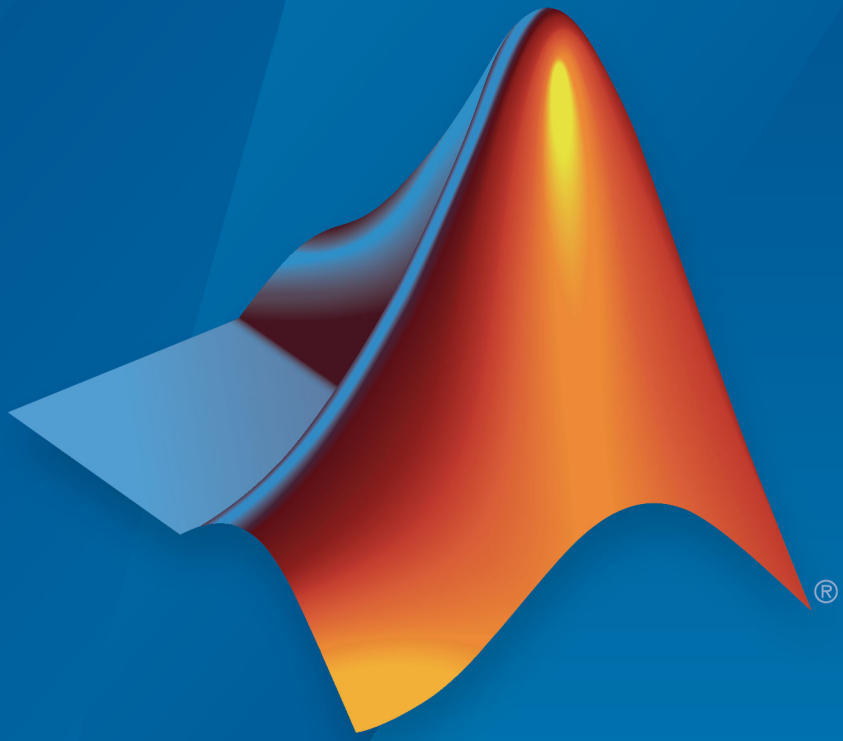
MATLAB® Coder™

Reference



MATLAB®

R2019a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

MATLAB® Coder™ Reference

Revision History

April 2011	Online only	New for Version 2 (R2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.4 (Release 2013a)
September 2013	Online only	Revised for Version 2.5 (Release 2013b)
March 2014	Online only	Revised for Version 2.6 (Release 2014a)
October 2014	Online only	Revised for Version 2.7 (Release 2014b)
March 2015	Online only	Revised for Version 2.8 (Release 2015a)
September 2015	Online only	Revised for Version 3.0 (Release 2015b)
October 2015	Online only	Rereleased for Version 2.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.1 (Release 2016a)
September 2016	Online only	Revised for Version 3.2 (Release 2016b)
March 2017	Online only	Revised for Version 3.3 (Release 2017a)
September 2017	Online only	Revised for Version 3.4 (Release 2017b)
March 2018	Online only	Revised for Version 4.0 (Release 2018a)
September 2018	Online only	Revised for Version 4.1 (Release 2018b)
March 2019	Online only	Revised for Version 4.2 (Release 2019a)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

Contents

Apps — Alphabetical List

MATLAB Coder

Generate C code or MEX function from MATLAB code

Description

The **MATLAB Coder** app generates C or C++ code from MATLAB code. You can generate:

- C or C++ source code, static libraries, dynamically linked libraries, and executables that you can integrate into existing C or C++ applications outside of MATLAB.
- MEX functions for accelerated versions of your MATLAB functions.

The workflow-based user interface steps you through the code generation process. Using the app, you can:

- Create a project or open an existing project. The project specifies the input files, entry-point function input types, and build configuration.
- Review code generation readiness issues, including unsupported functions.
- Check your MATLAB function for run-time issues.
- Fix issues in your MATLAB code using the integrated editor.
- Convert floating-point MATLAB code to fixed-point C code (requires a Fixed-Point Designer™ license).
- Convert double-precision MATLAB code to single-precision C code (requires a Fixed-Point Designer license).
- Trace from MATLAB code to generated C or C++ source code through comments.
- See static code metrics (requires an Embedded Coder® license).
- Verify the numerical behavior of generated code using software-in-the-loop and processor-in-the-loop execution (requires an Embedded Coder license).
- Export project settings in the form of a MATLAB script.
- Access generated files.
- Package generated files as a single zip file for deployment outside of MATLAB.

When the app creates a project, if the Embedded Coder product is installed, the app enables Embedded Coder features. When Embedded Coder features are enabled, code

generation requires an Embedded Coder license. To disable Embedded Coder features, in the project build settings, on the **All Settings** tab, under **Advanced**, set **Use Embedded Coder features** to `No`.

Open the MATLAB Coder App

- MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
- MATLAB command prompt: Enter `coder`.

Examples

- "Generate C Code by Using the MATLAB Coder App"

Programmatic Use

```
coder
```

See Also

Apps
Fixed-Point Converter

Functions
`codegen`

Topics
"Generate C Code by Using the MATLAB Coder App"

Function Reference

cnncodegen

Generate code and build static library for Series or DAG Network

Syntax

```
cnncodegen(net,'targetlib',libraryname)
cnncodegen(net,'targetlib',libraryname,Name,Value)
```

Description

`cnncodegen(net,'targetlib',libraryname)` generates C++ code and builds a static library for the specified network object and target library by using default values for all properties.

`cnncodegen(net,'targetlib',libraryname,Name,Value)` generates C++ code and builds a static library for the specified network object and target library with additional code generation options specified by one or more `Name,Value` pair arguments.

Examples

Generate C++ Code for a Pretrained Network to Run on an ARM Processor

Use `cnncodegen` to generate C++ code for a pretrained network for deployment to an ARM processor.

Get the pretrained AlexNet model by using the `alexnet` function. This function requires the Deep Learning Toolbox™ Model for AlexNet Network. If you have not installed this support package, the function provides a download link. Alternatively, see Deep Learning Toolbox Model *for AlexNet Network*.

```
net = alexnet;
```

Generate code by using `cnncodegen` with `'targetlib'` set to `'arm-compute'`. For `'arm-compute'`, you must provide the `'ArmArchitecture'` parameter.

```
cnncodegen(net,'targetlib','arm-compute','targetparams',struct('ArmComputeVersion','18.05','ArmArchitecture'
```

Generate Code for the YOLO Network to Run on NVIDIA GPU

Generate CUDA® C++ code from a `SeriesNetwork` object created for the YOLO architecture, trained for classifying the PASCAL dataset. This example requires the GPU Coder™ product and GPU Coder Interface for Deep Learning Libraries.

Get the pretrained YOLO network and convert it into a `SeriesNetwork` object.

```
url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/Yolo/yolonet.mat';
websave('yolonet.mat',url);
net = coder.loadDeepLearningNetwork('yolonet.mat');
```

The `SeriesNetwork` object `net` contains 58 layers. These layers are convolution layers followed by leaky `ReLU` and fully connected layers at the end of the network architecture. You can use `net.Layers` to see the all the layers in this network.

Use the `cnncodegen` function to generate CUDA code.

```
cnncodegen(net,'targetlib','cudnn');
```

The code generator generates the `.cu` and header files in the `'/pwd/codegen'` folder. The series network is generated as a C++ class called `CnnMain`, containing an array of 58 layer classes. The `setup()` method of this class sets up handles and allocates resources for each layer object. The `predict()` method invokes prediction for each of the 58 layers in the network. The `cleanup()` method releases all the memory and system resources allocated for each layer object. All the binary weights (`cnn_**_w`) and the bias files (`cnn_**_b`) for the convolution layers of the network are stored in the `codegen` folder. The files are compiled into the static library `cnnbuild.a` (on Linux®) or `cnnbuild.lib` (on Windows®).

Input Arguments

`net` — Name of the series or DAG network object

character vector

Pretrained `SeriesNetwork` or `DAGNetwork` object.

`libraryname` — Deep learning target library

character vector | string scalar

The target library and the target platform to generate code for, specified as one of the values in this table.

Value	Description
`'arm-compute'`	Target an ARM® CPU processor supporting `NEON` instructions by using the ARM Compute Library for computer vision and machine learning. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'cudnn'`	Target NVIDIA® GPUs by using the CUDA Deep Neural Network library (cuDNN). Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.
`'mkldnn'`	Target Intel® CPU processor by using the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'tensorrt'`	Target NVIDIA GPUs by using NVIDIA TensorRT, a high performance deep learning inference optimizer and run-time library. Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`cnncodegen(net,'targetlib','mkldnn','codegenonly',0,'batchsize',1)`
generates C++ code for the Intel processor by using MKL-DNN and builds a static library for the network object in `net`.

General Options

`batchsize` — Size
integer

A positive nonzero integer value specifying the number of observations to operate on in a single call to the network `predict()` method. When calling `network->predict()`, the size of the input data must match the `batchsize` value specified during `cnncodegen`.

If `libraryname` is `'arm-compute'`, the value of `batchsize` must be `1`.

`codegenonly` — Flag
0 (default) | 1

Boolean flag that, when enabled, generates CUDA code without generating and building a makefile.

`targetparams` — Library-specific parameters
structure

Library-specific parameters specified as a 1-by-1 structure containing the fields described in these tables.

Parameters for ARM Compute Library

Field	Description
`ArmComputeVersion`	Version of ARM Compute Library on the target hardware, specified as `'18.03'` or `'18.05'`. The default value is `'18.05'`. If you set `ArmComputeVersion` to a version later than `'18.05'`, `ArmComputeVersion` is set to `'18.05'`
`ArmArchitecture`	ARM architecture supported on the target hardware, specified as `'armv7` or `'armv8'`. The specified architecture must be the same as the architecture for the ARM Compute Library on the target hardware. `ArmArchitecture` is a required parameter.

Parameters for NVIDIA cuDNN Library

Field	Description
`AutoTuning`	Enable or disable auto tuning feature. Enabling auto tuning allows the cuDNN library to find the fastest convolution algorithms. This increases performance for larger networks such as SegNet and ResNet. Default value is `true`. **Note** If `AutoTuning` is enabled for `TensorRT` targets, the software generates code with auto tuning disabled. It does so without generating any warning or error messages.

Parameters for NVIDIA TensorRT Library

Field	Description
`DataType`	Specify the precision of the tensor data type input to the network or the tensor output of a layer. When performing inference in 32-bit floats, use `'FP32'`. For 8-bit integer, use `'INT8'`. For half-precision, use `'FP16'` Default value is `'FP32'`. The `computecapability` argument must be set to `'7.0'` or higher if the `DataType` is set to `'FP16'`. The `computecapability` argument must be set to `'6.1'` or higher if the `DataType` is set to `'INT8'`. Compute capability of 6.2 does not support `INT8` precision.
`DataPath`	Location of the image dataset used during recalibration. Default value is `''`. This option is applicable only when `DataType` is set to `'INT8'`. When you select the `'INT8'` option, TensorRT quantizes the floating-point data to `int8`. The recalibration is performed with a reduced set of the calibration data. The calibration data must be present in the image data location specified by `DataPath`.The images must be in folders whose names correspond to the labels for the images.
`NumCalibrationBatches`	Numeric value specifying the batch size for calibration. This value must not be greater than the number of images present in the image dataset. This option is applicable only when `DataType` is set to `'INT8'`.

GPU Options (GPU Coder Only)

`computecapability` — Compute version

`'3.5'` (default) | character vector

This property affects GPU targeting only.

Character vector specifying the NVIDIA GPU compute capability to compile for. Argument takes the format of `major#.minor#`. Possible values are `'3.2'|'3.5'|'3.7'|'5.0'|'5.2'|'5.3'|'6.0'|'6.1'|'6.2'|'7.0'|'7.1'|'7.2'`.

`targetarch` — Target architecture

'host' (default) | 'tk1' | 'tx1' | 'tx2'

This property affects GPU targeting only.

Specify the target architecture that you want to generate and compile CUDA code for.

Value	Description
`'host'`	Target the GPU device on the host machine.
`'tk1'`	Target the NVIDIA Tegra® Tk1 board. This option requires the `Linaro GCC` toolchain and is supported only on the Linux platform. For information on setting up the Linaro toolchain, see “Environment Variables” (GPU Coder).
`'tx1'`	Target the NVIDIA Tegra TX1 board. This option requires the `Linaro GCC` toolchain and is supported only on the Linux platform. For information on setting up the Linaro toolchain, see “Environment Variables” (GPU Coder).

Value	Description
`'tx2'`	Target the NVIDIA Tegra TX2 board. This option requires the `Linaro GCC` toolchain and is supported only on the Linux platform. For information on setting up the Linaro toolchain, see “Environment Variables” (GPU Coder).

See Also

`codegen` | `coder.loadDeepLearningNetwork`

Topics

“Code Generation for Deep Learning Networks with MKL-DNN”
“Deep Learning Prediction with ARM Compute Using cnncodegen”
“Code Generation for Object Detection Using YOLO v2” (GPU Coder)
“Deep Learning Prediction with NVIDIA TensorRT” (GPU Coder)
“Installing Prerequisite Products” (GPU Coder)
“Code Generation for Deep Learning Networks with cuDNN” (GPU Coder)
“Generated CNN Class Hierarchy” (GPU Coder)

Introduced in R2017b

codegen

Generate C/C++ code from MATLAB code

Syntax

```
codegen options function -args {func_inputs}
codegen options files function -args {func_inputs}
codegen options files function -args {func_inputs} -nargout
number_args
codegen options files function1 -args {func1_inputs} ... functionN -
args {funcN_inputs}

codegen project
```

Description

`codegen options function -args {func_inputs}` generates C or C++ code from a MATLAB function with inputs of type `func_inputs` and builds the generated code. Use the `options` argument to specify settings such as the code generation configuration object. The configuration object controls build type (MEX, lib, dll, or exe) and code generation parameters. For information on creating and using a configuration object, see "Configure Build Settings", `coder.config`, and the configuration object reference pages: `coder.CodeConfig`, `coder.MexCodeConfig`, and `coder.EmbeddedCodeConfig`.

If the function does not have inputs, omit the function-specific `-args {func_inputs}` option.

`codegen options files function -args {func_inputs}` generates C/C++ code from a MATLAB function that uses custom source code specified in external `files`. For more information, see "Call C/C++ Code from MATLAB Code" and "Configure Build for External C/C++ Code".

`codegen options files function -args {func_inputs} -nargout number_args` generates C/C++ code and controls the number of output arguments for the C/C++ function code generated from the MATLAB function. The files and options

arguments are optional. Use the `-nargout` option when not all of the MATLAB function outputs are needed. For more information, see “Specify Number of Entry-Point Function Input or Output Arguments to Generate”.

`codegen options files function1 -args {func1_inputs} ... functionN -args {funcN_inputs}` generates C/C++ code from multiple MATLAB functions. Write the input arguments separately for each function following the function name. You can also use the `-nargout` option for each function. The functions that you generate code from are called *entry-point functions*. For more information, see “Generate Code for Multiple Entry-Point Functions”.

`codegen project` generates code from a MATLAB Coder project file, for example, `test.prj`.

Examples

Generate a MEX Function from a MATLAB Function

Write a MATLAB function, `coderRand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1).

```
function r = coderRand() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
r = rand();
```

Generate and run a MEX function. Code generation defaults to MEX code generation when you do not specify a build target. By default, `codegen` names the generated MEX function `coderRand_mex`.

```
codegen coderRand
coderRand_mex
```

Generate C Static Library Files in a Custom Folder

Write a MATLAB function, `mcadd`, that returns the sum of two values.

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

Generate the C library files in a custom folder `mcaddlib` using the `-config:lib` option. Specify the first input type as a 1-by-4 vector of unsigned 16-bit integers. Specify the second input as a double-precision scalar.

```
codegen -d mcaddlib -config:lib  mcadd -args {zeros(1,4,'uint16'),0}
```

Generate an Executable

Write a MATLAB function, `coderRand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1).

```
function r = coderRand() %#codegen
r = rand();
```

Write a main C function, `c:\myfiles\main.c`, that calls `coderRand`.

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderRand.h"
#include "coderRand_initialize.h"
#include "coderRand_terminate.h"
int main()
{
    coderRand_initialize();
    printf("coderRand=%g\n", coderRand());
    coderRand_terminate();

    puts("Press enter to quit:");
    getchar();

    return 0;
}
```

Configure your code generation parameters to include the main C function, then generate the C executable.

```
cfg = coder.config('exe')
cfg.CustomSource = 'main.c'
cfg.CustomInclude = 'c:\myfiles'
codegen -config cfg coderRand
```

`codegen` generates a C executable, `coderRand.exe`, in the current folder, and supporting files in the default folder, `codegen\exe\coderRand`.

This example shows how to specify a main function as a parameter in the configuration object `coder.CodeConfig`. Alternatively, you can specify the file containing `main()` separately at the command line. You can use a source, object, or library file.

For a more detailed example, see “Use an Example C Main in an Application”.

Generate Code That Uses a Variable-Size Input

Write a MATLAB function that takes a single input.

```
function y = halfValue(vector) %codegen
    y = 0.5 * vector;
end
```

Use `coder.typeof` to define an input type as a row vector of doubles with a maximum size of 1-by-16, with the second dimension variable-size.

```
vectorType = coder.typeof(1, [1 16], [false true]);
```

Generate a C static library.

```
codegen -config:lib halfValue -args {vectorType}
```

Generate Code That Uses Global Data

Write a MATLAB function, `use_globals`, that takes one input parameter `u` and 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');
global AR;
global B;
AR(1) = u(1) + B(1);
y = AR * 2;
```

Generate a MEX function. By default, `codegen` generates a MEX function named `use_globals_mex` in the current folder. Specify the properties of the global variables at the command line by using the `-globals` option. Specify that input `u` is a real, scalar, double, by using the `-args` option.

```
codegen -globals {'AR', ones(4), 'B', [1 2 3 4]} use_globals -args {0}
```

Alternatively, you can initialize the global data in the MATLAB workspace. At the MATLAB prompt, enter:

```
global AR B;
AR = ones(4);
B = [1 2 3];
```

Generate the MEX function.

```
codegen use_globals -args {0}
```

Generate Code That Accepts an Enumerated Type Input

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

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

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

Define an enumeration `LEDColor`. On the MATLAB path, create a file named 'LEDColor' containing:

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

Create a `coder.EnumType` object using a value from an existing MATLAB enumeration.

Define an enumeration `sysMode`. On the MATLAB path, create a file named 'sysMode' containing:

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

Create a `coder.EnumType` object from this enumeration.

```
t = coder.typeof(sysMode.OFF);
```

Generate a MEX function for `displayState`.

```
codegen  displayState -args {t}
```

Generate a Static Library That Accepts a Fixed-Point Input

Write a MATLAB language function, `mcsqrtfi`, that computes the square root of a fixed-point input.

```
function y = mcsqrtfi(x) %#codegen
y = sqrt(x);
```

Define `numerictype` and `fimath` properties for the fixed-point input `x` and generate C library code for `mcsqrtfi` using the `-config:lib` option.

```
T = numerictype('WordLength',32, ...
                'FractionLength',23, ...
                'Signed',true)
F = fimath('SumMode','SpecifyPrecision', ...
           'SumWordLength',32, ...
           'SumFractionLength',23, ...
           'ProductMode','SpecifyPrecision', ...
           'ProductWordLength',32, ...
           'ProductFractionLength',23)
% Define a fixed-point variable with these
%  numerictype and fimath properties
```

```
myfiprops = {fi(4.0,T,F)}
codegen -config:lib mcsqrtfi -args myfiprops
```

`codegen` generates C library and supporting files in the default folder, `codegen/lib/mcsqrtfi`.

Convert Floating-Point MATLAB Code to Fixed-Point C Code

This example requires Fixed-Point Designer.

Write a MATLAB function, `myadd`, that returns the sum of two values.

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

Write a MATLAB function, `myadd_test`, to test `myadd`.

```
function y = myadd_test %#codegen
    y = myadd(10,20);
end
```

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name.

```
fixptcfg.TestBenchName = 'myadd_test';
```

Create a code generation configuration object to generate a standalone C static library.

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

Generate the code using the `-float2fixed` option.

```
codegen -float2fixed fixptcfg -config cfg myadd
```

Input Arguments

`options` — Code generation options
option value | space delimited list of option values

The `codegen` command gives precedence to individual command-line options over options specified by a configuration object. If command-line options conflict, the rightmost option prevails. The order of the options and the other syntax elements is interchangeable.

Specified as one or more of these values:

`-c`	Generate C/C++ code, but do not invoke the `make` command.
`-config:dll`	Generate a dynamic C/C++ library using the default configuration parameters.
`-config:exe`	Generate a static C/C++ executable using the default configuration parameters.
`-config:lib`	Generate a static C/C++ library using the default configuration parameters.
`-config:mex`	Generate a MEX function using the default configuration parameters.
`-config:single`	Generate single-precision MATLAB code using the default configuration parameters. Requires Fixed-Point Designer.

`-config` *`config_object`*

Specify the configuration object that contains the code generation parameters. *`config_object`* is one of the following configuration objects:

- `coder.CodeConfig` — Parameters for standalone C/C++ library or executable generation if Embedded Coder is not available.

  ```
  % Configuration object for a dynamic linked library
  cfg = coder.config('dll')
  % Configuration object for an executable
  cfg = coder.config('exe')
  % Configuration object for a static standalone library
  cfg = coder.config('lib')
  ```

- `coder.EmbeddedCodeConfig`— Parameters for a standalone C/C++ library or executable generation if Embedded Coder is available.

  ```
  % Configuration object for a dynamic linked library
  ec_cfg = coder.config('dll')
  % Configuration object for an executable
  ec_cfg = coder.config('exe')
  % Configuration object for a static standalone library
  ec_cfg = coder.config('lib')
  ```

- `coder.MexCodeConfig` — Parameters for MEX code generation.

  ```
  mex_cfg = coder.config
  % or
  mex_cfg = coder.config('mex')
  ```

For more information, see “Configure Build Settings”.

`-d out_folder`

Store generated files in the absolute or relative path specified by *`out_folder`*. *`out_folder`* must not contain:

- Spaces, as spaces can lead to code generation failures in certain operating system configurations.
- Non 7-bit ASCII characters, such as Japanese characters,

If the folder specified by *`out_folder`* does not exist, `codegen` creates it.

If you do not specify the folder location, `codegen` generates files in the default folder:

`codegen/`*`target`*`/`*`fcn_name`*`.`

`target` can be:

- `mex` for MEX functions
- `exe` for embeddable C/C++ executables
- `lib` for embeddable C/C++ libraries
- `dll` for C/C++ dynamic libraries

`fcn_name` is the name of the first MATLAB function (alphabetically) at the command line.

The function does not support the following characters in folder names: asterisk (*), question-mark (?), dollar ($), and pound (#).

Note Each time `codegen` generates the same type of output for the same code, it removes the files from the previous build. If you want to preserve files from a previous build, before starting another build, copy them to a different location.

`-double2single` *`double2single_cfg_name`*

Generates single-precision MATLAB code using the settings that the `coder.SingleConfig` object *`double2single_cfg_name`* specifies. `codegen` generates files in the folder `codegen/`*`fcn_name`*`/single`.

`fcn_name` is the name of the entry-point function.

When used with the `-config` option, also generates single-precision C/C++ code. `codegen` generates the single-precision files in the folder `codegen/`*`target`*`/`*`folder_name`*

. *`target`* can be:

- `mex` for MEX functions
- `exe` for embeddable C/C++ executables
- `lib` for embeddable C/C++ libraries
- `dll` for C/C++ dynamic libraries

`folder_name` is the concatenation of *`fcn_name`* and *`singlesuffix`*.

`singlesuffix` is the suffix that the `coder.SingleConfig` property `OutputFileNameSuffix` specifies. The single-precision files in this folder also have this suffix.

For more information, see “Generate Single-Precision MATLAB Code”. You must have Fixed-Point Designer to use this option.

`-float2fixed` *`float2fixed_cfg_name`*

When used with the `-config` option, generates fixed-point C/C++ code using the settings that the floating-point to fixed-point conversion configuration object *`float2fixed_cfg_name`* specifies.

`codegen` generates files in the folder `codegen/`*`target`*`/`*`fcn_name`*`_fixpt`. *`target`* can be:

- `mex` for MEX functions
- `exe` for embeddable C/C++ executables
- `lib` for embeddable C/C++ libraries
- `dll` for C/C++ dynamic libraries

`fcn_name` is the name of the entry-point function.

When used without the `-config` option, generates fixed-point MATLAB code using the settings that the floating-point to fixed-point conversion configuration object named *`float2fixed_cfg_name`* specifies. `codegen` generates files in the folder `codegen/`*`fcn_name`*`/fixpt`.

You must set the `TestBenchName` property of *`float2fixed_cfg_name`*. For example:

```
fixptcfg.TestBenchName = 'myadd_test';
```

This command specifies that `myadd_test` is the test file for the floating-point to fixed-point configuration object `fixptcfg`.

For more information, see “Convert MATLAB Code to Fixed-Point C Code”. You must have Fixed-Point Designer to use this option.

`-g`	Specify whether to use the debug option for the C compiler. If you enable debug mode, the C compiler disables some optimizations. The compilation is faster, but the execution is slower.

`-globals` *`global_values`*

Specify names and initial values for global variables in MATLAB files.

`global_values` is a cell array of global variable names and initial values. The format of `global_values` is:

```
{g1, init1, g2, init2, ..., gn, initn}
```

`gn` is the name of a global variable specified as a character vector. `initn` is the initial value. For example:

```
-globals {'g', 5}
```

Alternatively, use this format:

```
-globals {global_var, {type, initial_value}}
```

`type` is a type object. To create the type object, use `coder.typeof`. For global cell array variables, you must use this format.

Before generating code with `codegen`, initialize global variables. If you do not provide initial values for global variables using the `-globals` option, `codegen` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `codegen` generates an error.

MATLAB Coder and MATLAB each have their own copies of global data. For consistency, synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables can differ.

To specify a constant value for a global variable, use `coder.Constant`. For example:

```
-globals {'g', coder.Constant(v)}
```

	specifies that `g` is a global variable with constant value `v`. For more information, see “Generate Code for Global Data”.
`-I` *`include_path`*	Add *`include_path`* to the beginning of the code generation path. When `codegen` searches for MATLAB functions and custom C/C++ files, it searches the code generation path first. It does not search for classes on the code generation path. Classes must be on the MATLAB search path. For more information, see “Paths and File Infrastructure Setup”. If the path contains characters that are not 7-bit ASCII, such as Japanese characters, it is possible that `codegen` does not find files on this path. If your *`include_path`* contains paths that contain spaces, enclose each instance in double quotes, for example: `'C:\Project "C:\Custom Files"'`
`-jit`	Use just-in-time (JIT) compilation for generation of a MEX function. JIT compilation can speed up MEX function generation. This option applies only to MEX function generation. This option is not compatible with certain code generation features or options, such as custom code or using the OpenMP library.
`-launchreport`	Generate and open a code generation report. If you do not specify this option, `codegen` generates a report only if error or warning messages occur or if you specify the `-report` option.

`-o output_file_name`

Generate the MEX function, C/C++ library, or C/C++ executable file with the base name *`output_file_name`* plus an extension:

- `.a` or `.lib` for C/C++ static libraries
- `.exe` or no extension for C/C++ executables
- `.dll` for C/C++ dynamic libraries on Microsoft® Windows systems
- `.so` for C/C++ dynamic libraries on Linux systems
- `.dylib` for C/C++ dynamic libraries on Mac systems
- Platform-dependent extension for generated MEX functions

`output_file_name` can be a file name or include an existing path. *`output_file_name`* must not contain spaces, as spaces can lead to code generation failures in certain operating system configurations.

For MEX functions, *`output_file_name`* must be a valid MATLAB function name.

If you do not specify an output file name for libraries and executables, the base name is *`fcn_1`*. *`fcn_1`* is the name of the first MATLAB function specified at the command line. For MEX functions, the base name is *`fcn_1_mex`*. You can run the original MATLAB function and the MEX function and compare the results.

`-O` *`optimization_option`*	Optimize generated code, based on the value of *`optimization_option`*: • `enable:inline` — Enable function inlining • `disable:inline` — Disable function inlining • `enable:openmp` — Use OpenMP library if available. Using the OpenMP library, the MEX functions or C/C++ code that `codegen` generates for `parfor`-loops can run on multiple threads. • `disable:openmp` — Disable OpenMP library. With OpenMP disabled, `codegen` treats `parfor`-loops as `for`-loops and generates a MEX function or C/C++ code that runs on a single thread. See “Control Compilation of parfor-Loops”. Specify `-O` at the command line once for each optimization. If not specified, `codegen` uses inlining and OpenMP for optimization.
`-preservearraydims`	Generate code that uses N-dimensional indexing. For more information, see “Generate Code That Uses N-Dimensional Indexing”.
`-profile`	Enable profiling of generated MEX function by using the MATLAB Profiler. For more information, see “Profile MEX Functions by Using MATLAB Profiler”.
`-report`	Produce a code generation report. If you do not specify this option, `codegen` produces a report only if error or warning messages occur or if you specify the `-launchreport` option. If you have Embedded Coder, this option also enables production of the Code Replacements report.

`-reportinfo` *`info`*	Export information about code generation to the variable *`info`* in your base MATLAB workspace. See “Access Code Generation Report Information Programmatically”.
`-rowmajor`	Generate code that uses row-major array layout. Column-major layout is the default. For more information, see “Generate Code That Uses Row-Major Array Layout”.
`-singleC`	Generate single-precision C/C++ code. For more information, see “Generate Single-Precision C Code at the Command Line”. You must have Fixed-Point Designer to use this option.
`-test` *`test_file`*	Run *`test_file`*, replacing a call to the original MATLAB function with a call to the MEX function. Using this option is the same as running `coder.runTest`. This option is supported only when generating MEX functions or when using a configuration object with `VerificationMode` set to `'SIL'` or `'PIL'`. Creation of a configuration object that has the `VerificationMode` parameter requires the Embedded Coder product. This option is not supported with fixed-point conversion or single-precision conversion.
`-v`	Enable verbose mode to show build steps. Use when generating libraries or executables only.
`-?`	Display help for `codegen` command.

`function` — Name of MATLAB function to generate code from
function name

Specified as a function existing in the current working folder or on the path. If the MATLAB file is on a path that contains non 7-bit ASCII characters, such as Japanese characters, the `codegen` command might not find the file.

If you are using the LCC compiler, do not name an entry-point function `main`.

Example: `codegen myAddFunction`

`func_inputs` — Example values for MATLAB function inputs
expression | variable | literal value | `coder.Type` object

Example values that define the size, class, and complexity of the inputs of the preceding MATLAB function. The position of the input in the cell array must correspond to the position of the input argument in the MATLAB function definition. Alternatively, instead of an example value, you can provide a `coder.Type` object. To create a `coder.Type` object, use `coder.typeof`.

To generate a function that has fewer input arguments than the function definition has, omit the example values for the arguments that you do not want.

For more information, see “Specify Properties of Entry-Point Function Inputs”.

Example: `codegen foo -args {1}`

Example: `codegen foo2 -args {1, ones(3,5)}`

Example: `codegen foo3 -args {1, ones(3,5), coder.typeof("hello")}`

`files` — Names of custom source files
file name | space delimited list of file names

Space-separated list of custom files to include in generated code. The order of the options, external files, and function specifications is interchangeable. You can include these types of files:

- C file (`.c`)
- C++ file (`.cpp`)
- Header file (`.h`)
- Object file (`.o` or `.obj`)
- Library (`.a`, `.so`, `.dylib`, or `.lib`)
- Template makefile (`.tmf`)

Note Support for template makefiles (TMF) will be removed in a future release. Instead, use the toolchain approach for building the generated code.

If these files are on a path that contains non 7-bit ASCII characters, such as Japanese characters, the `codegen` command might not find the files.

Example: `codegen foo myLib.lib`

`number_args` — Number of output arguments in the generated entry-point function
integer

Number of output arguments in the C/C++ entry-point function generated for the preceding MATLAB function. The code generator produces the specified number of output arguments in the order in which they occur in the MATLAB function definition.

Example: `codegen myMLfnWithThreeOuts -nargout 2`

`project` — Project file name
file name

Project file created from the MATLAB Coder app. The code generator uses the project file to set entry-point functions, input type definitions, and other options. To open the app and create or modify a project file, use the `coder` function.

Example: `codegen foo.prj`

Limitations

- You cannot generate code for MATLAB scripts. To generate code, rewrite the script as a function.

Tips

- By default, code is generated in the folder `codegen/`*`target`*`/`*`function`*. MEX functions and executables are copied to the current working folder.
- To simplify your code generation process, you can write your code generation commands in a separate script. In the script, define your function input types and code generation options. To generate code, call the script.
- Each time `codegen` 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 previous build, before starting another build, copy the files to a different location.

- Use the `coder` function to open the MATLAB Coder app and create a MATLAB Coder project. The app provides a user interface that facilitates adding MATLAB files, defining input parameters, and specifying build parameters.
- You can call `codegen` by using function syntax. Specify the `codegen` arguments as character vectors or string scalars. For example:

  ```
  codegen('myfunction','-args',{2 3},'-report')
  ```
- To provide a string scalar as an input or to specify a `codegen` argument as a string scalar, use the function syntax. For example:

  ```
  codegen('myfunction','-args',"mystring",'-report')
  codegen("myfunction","-args","mystring","-report")
  ```

  Providing string scalar inputs to the command form of `codegen` can produce unexpected results. See “Command vs. Function Syntax” (MATLAB).
- To perform programmatic `codegen` calls, use the function syntax. For example:

  ```
  A = {'myfunction','-args',{2 3}};
  codegen(A{:})
  ```

See Also

`coder` | `coder.EnumType` | `coder.FixptConfig` | `coder.config` | `coder.runTest` | `coder.typeof` | `fi` | `fimath` | `mex` | `numerictype` | `parfor`

Topics

“Generate C Code at the Command Line”
“Generate C Code by Using the MATLAB Coder App”
“Build Process Customization”

Introduced in R2011a

coder

Open MATLAB Coder app

Syntax

```
coder
coder projectname
coder -open projectname
coder -build projectname
coder -new projectname
coder -ecoder false -new projectname
coder -tocode projectname -script scriptname
coder -tocode projectname
```

Description

`coder` opens the MATLAB Coder app. To create a project, on the **Select Source Files** page, provide the entry-point file names. The app creates a project with a default name that is the name of the first entry-point file. To open an existing project, on the app toolbar, click  , and then click **Open existing project**.

If the Embedded Coder product is installed, when the app creates a project, it enables Embedded Coder features. When Embedded Coder features are enabled, code generation requires an Embedded Coder license. To disable Embedded Coder features, in the project build settings, on the **All Settings** tab, under **Advanced**, set **Use Embedded Coder features** to `No`.

`coder projectname` opens the MATLAB Coder app using the existing project named `projectname.prj`.

`coder -open projectname` opens the MATLAB Coder app using the existing project named `projectname.prj`.

`coder -build projectname` builds the existing project named `projectname.prj`.

`coder -new projectname` opens the MATLAB Coder app creating a project named `projectname.prj`. If the Embedded Coder product is installed, the app creates the project with Embedded Coder features enabled. To disable these features, in the project build settings, on the **All Settings** tab, under **Advanced**, set **Use Embedded Coder features** to `No`.

`coder -ecoder false -new projectname` opens the MATLAB Coder app creating a project named `projectname.prj`. The app creates the project with Embedded Coder features disabled even if the Embedded Coder product is installed.

`coder -tocode projectname -script scriptname` converts the existing project named `projectname.prj` to the equivalent script of MATLAB commands. The script is named `scriptname`.

- If `scriptname` exists, `coder` overwrites it.
- The script reproduces the project build configuration in a configuration object and builds the project. The script:
  - Creates a configuration object named `cfg`.
  - Defines the variable `ARGS` for function input types.
  - Defines the variable `GLOBALS` for global data initial values.
  - Runs the `codegen` command. When you run the script, the entry-point functions that are arguments to `codegen` must be on the search path.
- `cfg`, `ARGS`, and `GLOBALS` appear in the base workspace only after you run the script.

If the project includes automated fixed-point conversion, `coder` generates two scripts:

- A script `scriptname` that contains the MATLAB commands to:
  - Create a code configuration object that has the same settings as the project.
  - Run the `codegen` command to convert the fixed-point MATLAB function to a fixed-point C function.
- A script whose file name is a concatenation of the name specified by `scriptname` and the generated fixed-point file name suffix specified by the project file. If `scriptname` specifies a file extension, the script file name includes the file extension. For example, if `scriptname` is `myscript.m` and the suffix is the default value `_fixpt`, the script name is `myscript_fixpt.m`.

  This script contains the MATLAB commands to:

- Create a floating-point to fixed-point conversion configuration object that has the same fixed-point conversion settings as the project.
- Run the `codegen` command to convert the floating-point MATLAB function to a fixed-point MATLAB function.

For a project that includes fixed-point conversion, before converting the project to scripts, complete the **Test Numerics** step of the fixed-point conversion process.

`coder -tocode projectname` converts the existing project named `projectname.prj` to the equivalent script of MATLAB commands. It writes the script to the Command Window.

Examples

Open an existing MATLAB Coder project

Open the MATLAB Coder app using the existing MATLAB Coder project named `my_coder_project`.

```
coder -open my_coder_project
```

Build a MATLAB Coder project

Build the MATLAB Coder project named `my_coder_project`.

```
coder -build my_coder_project
```

Create a MATLAB Coder project

Open the MATLAB Coder app and create a project named `my_coder_project`.

```
coder -new my_coder_project
```

Convert a MATLAB Coder project to a MATLAB script

Convert the MATLAB Coder project named `my_coder_project.prj` to the MATLAB script named `myscript.m`.

```
coder -tocode my_coder_project -script my_script.m
```

Input Arguments

`projectname` — Name of MATLAB Coder project

character vector

Name of MATLAB Coder project that you want to create, open, or build. The project name must not contain spaces.

`scriptname` — Name of script file

character vector

Name of script that you want to create when using the `-tocode` option with the `-script` option. The script name must not contain spaces.

Tips

- If you are sharing an Embedded Coder license, use `coder -ecoder false -new projectname` to create a project that does not require this license. If the Embedded Coder product is installed, the app creates the project with Embedded Coder features disabled. When these features are disabled, code generation does not require an Embedded Coder license. To enable Embedded Coder features, in the project build settings, on the **All Settings** tab, under **Advanced**, set **Use Embedded Coder features** to `Yes`.
- Creating a project or opening an existing project causes other MATLAB Coder or Fixed-Point Converter projects to close.
- If your installation does not include the Embedded Coder product, the Embedded Coder settings do not show. However, values for these settings are saved in the project file. If you open the project in an installation that includes the Embedded Coder product, you see these settings.
- A Fixed-Point Converter project opens in the Fixed-Point Converter app. To convert the project to a MATLAB Coder project, in the Fixed-Point Converter app:

1. Click  and select **Reopen project as**.
2. Select `MATLAB Coder`.

Alternatives

- On the **Apps** tab, in the **Code Generation** section, click **MATLAB Coder**.
- Use the `codegen` function to generate code at the command line.

See Also

MATLAB Coder | `codegen`

Topics

“Generate C Code by Using the MATLAB Coder App”
“Convert MATLAB Coder Project to MATLAB Script”
“Convert Fixed-Point Conversion Project to MATLAB Scripts”
“Convert MATLAB Code to Fixed-Point C Code”

Introduced in R2011a

coder.allowpcode

Package: coder

Control code generation from protected MATLAB files

Syntax

```
coder.allowpcode('plain')
```

Description

`coder.allowpcode('plain')` allows you to generate protected MATLAB code (P-code) that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as protected P-files that provide code generation optimizations, providing intellectual property protection for your source MATLAB code.

Call this function in the top-level function before control-flow statements, such as `if`, `while`, `switch`, and function calls.

MATLAB functions can call P-code. When the `.m` and `.p` versions of a file exist in the same folder, the P-file takes precedence.

`coder.allowpcode` is ignored outside of code generation.

Examples

Generate optimized embeddable code from protected MATLAB code:

1. Write an function `p_abs` that returns the absolute value of its input:

```
function out = p_abs(in)   %#codegen
% The directive %#codegen indicates that the function
```

```
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);
```

2. Generate protected P-code. At the MATLAB prompt, enter:

   ```
   pcode p_abs
   ```

   The P-file, `p_abs.p`, appears in the current folder.

3. Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license). At the MATLAB prompt, enter:

   ```
   codegen p_abs -args { int32(0) }
   ```

   `codegen` generates a MEX function in the current folder.

4. Generate embeddable C code for `p_abs.p` (requires a MATLAB Coder license). At the MATLAB prompt, enter:

   ```
   codegen p_abs -config:lib -args { int32(0) };
   ```

   `codegen` generates C library code in the `codegen\lib\p_abs` folder.

See Also

`codegen` | `pcode`

Topics

“Compilation Directive %#codegen”

Introduced in R2011a

coder.approximation

Create function replacement configuration object

Syntax

```
q = coder.approximation(function_name)
q = coder.approximation('Function',function_name,Name,Value)
```

Description

`q = coder.approximation(function_name)` creates a function replacement configuration object for use during code generation or fixed-point conversion. The configuration object specifies how to create a lookup table approximation for the MATLAB function specified by `function_name`. To associate this approximation with a `coder.FixptConfig` object for use with the`codegen` function, use the `coder.FixptConfig` configuration object `addApproximation` method.

Use this syntax only for the functions that `coder.approximation` can replace automatically. These functions are listed in the `function_name` argument description.

`q = coder.approximation('Function',function_name,Name,Value)` creates a function replacement configuration object using additional options specified by one or more name-value pair arguments.

Examples

Replace `log` Function with Default Lookup Table

Create a function replacement configuration object using the default settings. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('log');
```

Replace `log` Function with Uniform Lookup Table

Create a function replacement configuration object. Specify the input range and prefix to add to the replacement function name. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('Function','log','InputRange',[0.1,1000],...
'FunctionNamePrefix','log_replace_');
```

Replace `log` Function with Optimized Lookup Table

Create a function replacement configuration object using the `'OptimizeLUTSize'` option to specify to replace the `log` function with an optimized lookup table. The resulting lookup table in the generated code uses less than the default number of points.

```
    logAppx = coder.approximation('Function','log','OptimizeLUTSize', true,...
'InputRange',[0.1,1000],'InterpolationDegree',1,'ErrorThreshold',1e-3,...
'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Replace Custom Function with Optimized Lookup Table

Create a function replacement configuration object that specifies to replace the custom function, `saturateExp`, with an optimized lookup table.

Create a custom function, `saturateExp`.

```
saturateExp = @(x) 1/(1+exp(-x));
```

Create a function replacement configuration object that specifies to replace the `saturateExp` function with an optimized lookup table. Because the `saturateExp` function is not listed as a function for which `coder.approximation` can generate an approximation automatically, you must specify the `CandidateFunction` property.

```
saturateExp = @(x) 1/(1+exp(-x));
custAppx = coder.approximation('Function','saturateExp',...
```

```
'CandidateFunction', saturateExp,...
'NumberOfPoints',50,'InputRange',[0,10]);
```

Input Arguments

`function_name` — Name of the function to replace

`'acos'` | `'acosd'` | `'acosh'` | `'acoth'` | `'asin'` | `'asind'` | `'asinh'` | `'atan'` | `'atand'` | `'atanh'` | `'cos'` | `'cosd'` | `'cosh'` | `'erf '` | `'erfc'` | `'exp'` | `'log'` | `'normcdf'` | `'reallog'` | `'realsqrt'` | `'reciprocal'` | `'rsqrt'` | `'sin'` | `'sinc'` | `'sind'` | `'sinh'` | `'sqrt'` | `'tan'` | `'tand'`

Name of function to replace, specified as a string. The function must be one of the listed functions.

Example: `'sqrt'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Function', 'log'`

`Architecture` — Architecture of lookup table approximation

`'LookupTable'` (default) | `'Flat'`

Architecture of the lookup table approximation, specified as the comma-separated pair consisting of `'Architecture'` and a string. Use this argument when you want to specify the architecture for the lookup table. The `Flat` architecture does not use interpolation.

Data Types: `char`

`CandidateFunction` — Function handle of the replacement function

function handle | string

Function handle of the replacement function, specified as the comma-separated pair consisting of `'CandidateFunction'` and a function handle or string referring to a function handle. Use this argument when the function that you want to replace is not

listed under `function_name`. Specify the function handle or string referring to a function handle of the function that you want to replace. You can define the function in a file or as an anonymous function.

If you do not specify a candidate function, then the function you chose to replace using the `Function` property is set as the `CandidateFunction`.

Example: `'CandidateFunction', @(x) (1./(1+x))`

Data Types: `function_handle` | `char`

`ErrorThreshold` — Error threshold value used to calculate optimal lookup table size
0.001 (default) | nonnegative scalar

Error threshold value used to calculate optimal lookup table size, specified as the comma-separated pair consisting of `'ErrorThreshold'` and a nonnegative scalar. If `'OptimizeLUTSize'` is `true`, this argument is required.

`Function` — Name of function to replace with a lookup table approximation
`function_name`

Name of function to replace with a lookup table approximation, specified as the comma-separated pair consisting of `'Function'` and a string. The function must be continuous and stateless. If you specify one of the functions that is listed under `function_name`, the conversion process automatically provides a replacement function. Otherwise, you must also specify the `'CandidateFunction'` argument for the function that you want to replace.

Example: `'Function','log'`

Example: `'Function', 'my_log','CandidateFunction',@my_log`

Data Types: `char`

`FunctionNamePrefix` — Prefix for generated fixed-point function names
'replacement_' (default) | string

Prefix for generated fixed-point function names, specified as the comma-separated pair consisting of `'FunctionNamePrefix'` and a string. The name of a generated function consists of this prefix, followed by the original MATLAB function name.

Example: 'log_replace_'

`InputRange` — Range over which to replace the function
`[ ]` (default) | 2x1 row vector | 2xN matrix

Range over which to replace the function, specified as the comma-separated pair consisting of `'InputRange'` and a 2-by-1 row vector or a 2-by-*N* matrix.

Example: `[-1 1]`

`InterpolationDegree` — Interpolation degree
1 (default) | `0` | `2` | `3`

Interpolation degree, specified as the comma-separated pair consisting of `'InterpolationDegree'` and`1` (linear), `0` (none), `2` (quadratic), or `3` (cubic).

`NumberOfPoints` — Number of points in lookup table
`1000` (default) | positive integer

Number of points in lookup table, specified as the comma-separated pair consisting of `'NumberOfPoints'` and a positive integer.

`OptimizeIterations` — Number of iterations
25 (default) | positive integer

Number of iterations to run when optimizing the size of the lookup table, specified as the comma-separated pair consisting of `'OptimizeIterations'` and a positive integer.

`OptimizeLUTSize` — Optimize lookup table size
`false` (default) | `true`

Optimize lookup table size, specified as the comma-separated pair consisting of `'OptimizeLUTSize'` and a logical value. Setting this property to `true` generates an area-optimal lookup table, that is, the lookup table with the minimum possible number of points. This lookup table is optimized for size, but might not be speed efficient.

`PipelinedArchitecture` — Option to enable pipelining
`false` (default) | `true`

Option to enable pipelining, specified as the comma-separated pair consisting of `'PipelinedArchitecture'` and a logical value.

Output Arguments

q — Function replacement configuration object, returned as a `coder.mathfcngenerator.LookupTable` or a `coder.mathfcngenerator.Flat` configuration object

`coder.mathfcngenerator.LookupTable` configuration object | `coder.mathfcngenerator.Flat` configuration object

Function replacement configuration object. Use the `coder.FixptConfig` configuration object `addApproximation` method to associate this configuration object with a `coder.FixptConfig` object. Then use the `codegen` function `-float2fixed` option with `coder.FixptConfig` to convert floating-point MATLAB code to fixed-point code.

Property	Default Value
`Auto-replace function`	`''`
`InputRange`	`[]`
`FunctionNamePrefix`	`'replacement_'`
`Architecture`	`LookupTable` (read only)
`NumberOfPoints`	`1000`
`InterpolationDegree`	`1`
`ErrorThreshold`	`0.001`
`OptimizeLUTSize`	`false`
`OptimizeIterations`	`25`

See Also

Classes
`coder.FixptConfig`

Functions
`codegen`

Topics

“Replace the exp Function with a Lookup Table”
“Replace a Custom Function with a Lookup Table”

“Replacing Functions Using Lookup Table Approximations”

Introduced in R2014b

coder.ceval

Call external C/C++ function

Syntax

```
coder.ceval(cfun_name)
coder.ceval(cfun_name,cfun_arguments)

coder.ceval('-global',cfun_name)
coder.ceval('-global',cfun_name,cfun_arguments)

coder.ceval('-layout:rowMajor',cfun_name,cfun_arguments)
coder.ceval('-layout:columnMajor',cfun_name,cfun_arguments)
coder.ceval('-layout:any',cfun_name,cfun_arguments)

cfun_return = coder.ceval( ___ )
```

Description

`coder.ceval(cfun_name)` executes the external C/C++ function specified by `cfun_name`. Define `cfun_name` in an external C/C++ source file or library. Provide the external source, library, and header files to the code generator.

`coder.ceval(cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments`. `cfun_arguments` is a comma-separated list of input arguments in the order that `cfun_name` requires.

By default, `coder.ceval` passes arguments by value to the C/C++ function whenever C/C++ supports passing arguments by value. To make `coder.ceval` pass arguments by reference, use the constructs `coder.ref`, `coder.rref`, and `coder.wref`. If C/C++ does not support passing arguments by value, for example, if the argument is an array, `coder.ceval` passes arguments by reference. If you do not use `coder.ref`, `coder.rref` or `coder.wref`, a copy of the argument can appear in the generated code to enforce MATLAB semantics for arrays.

`coder.ceval('-global',cfun_name)` executes `cfun_name` and indicates that `cfun_name` uses one or more MATLAB global variables. The code generator can then produce code that is consistent with this global variable usage.

`coder.ceval('-global',cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments` and indicates that `cfun_name` uses one or more MATLAB global variables.

`coder.ceval('-layout:rowMajor',cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments` and passes data stored in row-major layout. When called from a function that uses column-major layout, the code generator converts inputs to row-major layout and converts outputs back to column-major layout. For a shorter syntax, use `coder.ceval('-row',...)`.

`coder.ceval('-layout:columnMajor',cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments` and passes data stored in column-major layout. When called from a function that uses row-major layout, the code generator converts inputs to column-major layout and converts outputs back to row-major layout. For a shorter syntax, use `coder.ceval('-col',...)`.

`coder.ceval('-layout:any',cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments` and passes data with its current array layout, even when array layouts do not match. The code generator does not convert the array layout of the input or output data.

`cfun_return = coder.ceval( ___ )` executes `cfun_name` and returns a single scalar value, `cfun_return`, corresponding to the value that the C/C++ function returns in the `return` statement. To be consistent with C/C++, `coder.ceval` can return only a scalar value. It cannot return an array. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Call External C Function

Call a C function `foo(u)` from a MATLAB function from which you intend to generate C code.

Create a C header file `foo.h` for a function `foo` that takes two input parameters of type `double` and returns a value of type `double`.

```
double foo(double in1, double in2);
```

Write the C function `foo.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double in1, double in2)
{
  return in1 + in2;
}
```

Write a function `callfoo` that calls `foo` by using `coder.ceval`. Provide the source and header files to the code generator in the function.

```
function y = callfoo  %#codegen
y = 0.0;
if coder.target('MATLAB')
    % Executing in MATLAB, call MATLAB equivalent of
    % C function foo
    y = 10 + 20;
else
    % Executing in generated code, call C function foo
    coder.updateBuildInfo('addSourceFiles','foo.c');
    coder.cinclude('foo.h');
    y = coder.ceval('foo', 10, 20);
end
end
```

Generate C library code for function `callfoo`. The `codegen` function generates C code in the `\codegen\lib\callfoo` subfolder.

```
codegen -config:lib callfoo -report
```

Call a C Library Function

Call a C library function from MATLAB code.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)
%#codegen
y = abs(u);
```

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```
function y = callmyabsval(y)
%#codegen
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
  % Executing in MATLAB, call function myabsval
  y = myabsval(y);
else
  % add the required include statements to generated function code
  coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
  coder.cinclude('myabsval_initialize.h');
  coder.cinclude('myabsval.h');
  coder.cinclude('myabsval_terminate.h');

  % Executing in the generated code.
  % Call the initialize function before calling the
  % C function for the first time
  coder.ceval('myabsval_initialize');

  % Call the generated C library function myabsval
  y = coder.ceval('myabsval',y);

  % Call the terminate function after
  % calling the C function for the last time
  coder.ceval('myabsval_terminate');
end
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use `coder.updateBuildInfo` to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsva
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex(-2.75)
```

```
ans =

    2.7500
```

Call the MATLAB function `callmyabsval`.

```
callmyabsval(-2.75)
```

```
ans =

    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

Call C Function That Uses Global Variable

Use the `'-global'` flag when you call a C function that modifies a global variable.

Write a MATLAB function `useGlobal` that calls a C function `addGlobal`. Use the `'-global'` flag to indicate to the code generator that the C function uses a global variable.

```
function y = useGlobal()
global g;
t = g;
% compare execution with/without '-global' flag
coder.ceval('-global','addGlobal');
y = t;
end
```

Create a C header file `addGlobal.h` for the function `addGlobal`.

```
void addGlobal(void);
```

Write the C function `addGlobal` in the file `addGlobal.c`. This function includes the header file `useGlobal_data.h` that the code generator creates when you generate code for the function `useGlobal`. This header file contains the global variable declaration for `g`.

```
#include "addGlobal.h"
#include "useGlobal_data.h"
void addGlobal(void) {
    g++;
}
```

Generate the MEX function for `useGlobal`. To define the input to the code generator, declare the global variable in the workspace.

```
global g;
g = 1;
codegen useGlobal -report addGlobal.h addGlobal.c
y = useGlobal_mex();
```

With the `'-global'` flag, the MEX function produces the result `y = 1`. The `'-global'` flag indicates to the code generator that the C function possibly modifies the global variable. For `useGlobal`, the code generator produces this code:

```
real_T useGlobal(const emlrtStack *sp)
{
  real_T y;
  (void)sp;
  y = g;
  addGlobal();
  return y;
}
```

Without the `'-global'` flag, the MEX function produces `y = 2`. Because there is no indication that the C function modifies `g`, the code generator assumes that `y` and `g` are identical. This C code is generated:

```
real_T useGlobal(const emlrtStack *sp)
{
  (void)sp;
  addGlobal();
```

```
  return g;
}
```

Call C Function That Uses Different Array Layout

Suppose that you have a C function `testRM` that is designed to use row-major layout. You want to integrate this function into a MATLAB function `bar` that operates on arrays. The function `bar` is designed to use column-major layout, employing the `coder.columnMajor` directive.

```
function out = bar(in)
%#codegen
coder.columnMajor;
coder.ceval('-layout:rowMajor','testRM', ...
    coder.rref(in),coder.wref(out));
end
```

In the generated code, the code generator inserts a layout conversion from column-major layout to row-major layout on the variable `in` before passing it to `testRM`. On the output variable `out`, the code generator inserts a layout conversion back to column-major.

In general, if you do not specify the `layout` option for `coder.ceval`, the external function arguments are assumed to use column-major.

Input Arguments

`cfun_name` — C/C++ function name

character vector | string scalar

Name of external C/C++ function to call.

Example: `coder.ceval('foo')`

Data Types: `char` | `string`

`cfun_arguments` — C/C++ function arguments

scalar variable | array | element of an array | structure | structure field | object property

Comma-separated list of input arguments in the order that `cfun_name` requires.

Example: `coder.ceval('foo', 10, 20);`

Example: `coder.ceval('myFunction', coder.ref(x));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`
Complex Number Support: Yes

Limitations

- You cannot use `coder.ceval` on functions that you declare extrinsic with `coder.extrinsic`.
- When the LCC compiler creates a library, it adds a leading underscore to the library function names. If the compiler for the library was LCC and your code generation compiler is not LCC, you must add the leading underscore to the function name, for example, `coder.ceval('_mylibfun')`. If the compiler for a library was not LCC, you cannot use LCC to generate code from MATLAB code that calls functions from that library. Those library function names do not have the leading underscore that the LCC compiler requires.
- If a property has a get method, a set method, or validators, or is a System object™ property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- For code generation, before calling `coder.ceval`, you must specify the type, size, and complexity data type of return values and output arguments.
- To apply `coder.ceval` to a function that accepts or returns variables that do not exist in MATLAB code, such as pointers, `FILE` types for file I/O, and C/C++ macros, use the `coder.opaque` function.
- Use `coder.ceval` only in MATLAB for code generation. `coder.ceval` generates an error in uncompiled MATLAB code. To determine if a MATLAB function is executing in MATLAB, use `coder.target`. If the function is executing in MATLAB, call the MATLAB version of the C/C++ function.

See Also

`codegen` | `coder.ExternalDependency` | `coder.columnMajor` | `coder.extrinsic` | `coder.opaque` | `coder.ref` | `coder.rowMajor` | `coder.rref` | `coder.target` | `coder.updateBuildInfo` | `coder.wref`

Topics

“External Code Integration”
“Using C/C++ Code That MATLAB Coder Generates”
“Generate Code That Uses Row-Major Array Layout”
“Unknown Output Type for coder.ceval”

Introduced in R2011a

coder.cinclude

Include header file in generated code

Syntax

```
coder.cinclude(headerfile)
coder.cinclude(headerfile,'InAllSourceFiles',allfiles)
```

Description

`coder.cinclude(headerfile)` includes a header file in generated C/C++ source code.

MATLAB Coder generates the include statement in the C/C++ source files that are generated from the MATLAB code that contains the `coder.cinclude` call.

In a Simulink® model, when a `coder.cinclude` call appears in a MATLAB Function block, the code generator puts the include statement in the model header file.

`coder.cinclude(headerfile,'InAllSourceFiles',allfiles)` uses the `allfiles` option to determine whether to include the header file in almost all C/C++ source files.

If `allfiles` is `true`, MATLAB Coder generates the include statement in almost all C/C++ source files, except for some utility files. This behavior is the `coder.cinclude` behavior from R2016a and earlier releases. The presence of the include statement in these additional files can increase compile time and make the generated code less readable. Use this option only if your code depends on the legacy behavior. If `allfiles` is `false`, the behavior is the same as the behavior of `coder.cinclude(headerfile)`.

In a MATLAB Function block, `coder.cinclude(headerfile,'InAllSourceFiles', allfiles)` is the same as `coder.cinclude(headerfile)`.

Examples

Include Header File in C/C++ Code Generated by Using the MATLAB Coder codegen Command

Generate code from a MATLAB function that calls an external C function. Use `coder.cinclude` to include the required header file in the generated C code.

In a writable folder, create a subfolder `mycfiles`.

Write a C function `myMult2.c` that doubles its input. Save it in `mycfiles`.

```
#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
}
```

Write the header file `myMult2.h`. Save it in `mycfiles`.

```
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

Write a MATLAB function, `myfunc`, that includes `myMult2.h` and calls `myMult2` for code generation only.

```
function y = myfunc
%#codegen
y = 21;
if ~coder.target('MATLAB')
    % Running in generated code
    coder.cinclude('myMult2.h');
    y = coder.ceval('myMult2', y);
else
    % Running in MATLAB
    y = y * 2;
end
end
```

Create a code configuration object for a static library. Specify the locations of `myMult2.h` and `myMult2.c`

```
cfg = coder.config('lib');
cfg.CustomInclude = fullfile(pwd,'mycfiles');
cfg.CustomSource = fullfile(pwd,'mycfiles','myMult2.c');
```

Generate the code.

```
codegen -config cfg myfunc -report
```

The file `myfunc.c` contains this statement:

```
#include "myMult2.h"
```

The include statement does not appear in any other file.

Include Header File in C/C++ Code Generated from a MATLAB Function Block in a Simulink Model

Generate code from a MATLAB Function block that calls an external C function. Use `coder.cinclude` to include the required header file in the generated C code.

In a writable folder, create a subfolder `mycfiles`.

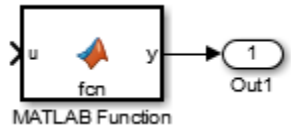
Write a C function `myMult2.c` that doubles its input. Save it in `mycfiles`.

```
#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
}
```

Write the header file `myMult2.h`. Save it in `mycfiles`.

```
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

Create a Simulink model that contains a MATLAB Function block connected to an Outport block.



In the MATLAB Function block, add the function `myfunc` that includes `myMult2.h` and calls `myMult2`.

```
function y = myfunc
%#codegen
y = 21;
coder.cinclude('myMult2.h');
y = coder.ceval('myMult2', y);
% Specify the locations of myMult2.h and myMult2.c
coder.extrinsic('pwd', 'fullfile');
customDir = coder.const(fullfile(pwd, 'mycfiles'));
coder.updateBuildInfo('addIncludePaths', customDir);
coder.updateBuildInfo('addSourcePaths', customDir);
coder.updateBuildInfo('addSourceFiles', 'myMult2.c');
end
```

Open the Configuration Parameters dialog box.

On the **Solver** pane, select a fixed-step solver.

Save the model as `mymodel`.

Build the model.

The file `mymodel.h` contains this statement:

```
#include "myMult2.h"
```

To read more about integrating custom code in a MATLAB Function block, see “Integrate C Code Using the MATLAB Function Block” (Simulink).

Input Arguments

`headerfile` — Name of header file

character vector | string scalar

Name of a header file specified as a character vector or string scalar. `headerfile` must be a compile-time constant.

Enclose a system header file name in angle brackets `< >`. The generated `#include` statement for a system header file has the format `#include <sysheader>`. A system header file must be in a standard location or on the include path. Specify the include path by using code generation custom code parameters.

Example: `coder.cinclude('<sysheader.h>')`

For a header file that is not a system header file, omit the angle brackets. The generated `#include` statement for a header file that is not a system header file has the format `#include "myHeader"`. The header file must be in the current folder or on the include path. Specify the include path by using code generation custom code parameters.

Example: `coder.cinclude('myheader.h')`

Data Types: `char`

`allfiles` — All source files option
true | false

Option to include header file in all generated C/C++ source files. If `allfiles` is `true`, MATLAB Coder generates the include statement in almost all of the C/C++ source files, except for some utility files. If `allfiles` is `false`, the behavior is the same as the behavior of `coder.cinclude(headerfile)`.

In a MATLAB Function block, the code generator ignores the all source files option.

Data Types: `logical`

Limitations

- Do not call `coder.cinclude` inside run-time conditional constructs such as `if` statements, `switch` statements, `while`-loops, and `for`-loops. You can call `coder.cinclude` inside compile-time conditional statements, such as `coder.target`. For example:

```
...
 if ~coder.target('MATLAB')
   coder.cinclude('foo.h');
   coder.ceval('foo');
end
...
```

Tips

- Before a `coder.ceval` call, call `coder.cinclude` to include the header file required by the external function that `coder.ceval` calls.

- Extraneous include statements in generated C/C++ code can increase compile time and reduce code readability. To avoid extraneous include statements in code generated by MATLAB Coder, follow these best practices:

  - Place a `coder.cinclude` call as close as possible to the `coder.ceval` call that requires the header file.
  - Do not set `allfiles` to `true`.

  For the MATLAB Function block, the code generator generates the include statement in the model header file.
- In R2016a and earlier releases, for any `coder.cinclude` call, MATLAB Coder included the header file in almost all generated C/C++ source files, except for some utility files. If you have code that depends on this legacy behavior, you can preserve the legacy behavior by using this syntax:

  ```
  coder.cinclude(headerfile,'InAllSourceFiles',true)
  ```

See Also

`codegen` | `coder.ceval` | `coder.config` | `coder.target`

Topics

“Configure Build for External C/C++ Code”

Introduced in R2013a

coder.columnMajor

Specify column-major array layout for a function or class

Syntax

```
coder.columnMajor
```

Description

`coder.columnMajor` specifies column-major array layout for the data used by the current function in generated code. When placed in a class constructor, `coder.columnMajor` specifies column-major layout for data used by the class.

Examples

Specify Column-Major Array Layout for a Function

Specify column-major array layout for a function by inserting `coder.columnMajor` into the function body.

Suppose that `myFunction` is the top-level function of your code. Your application requires you to perform matrix addition with column-major array layout and matrix multiplication with row-major layout.

```
function S = myFunction(A,B)
%#codegen
% check to make sure inputs are valid
if size(A,1) ~= size(B,1) || size(A,2) ~= size(B,2)
    disp('Matrices must be same size.')
    return;
end
% make both matrices symmetric
B = B*B';
A = A*A';
```

```
% add matrices
S = addMatrix(A,B);
end
```

Write a function for matrix addition called `addMatrix`. Specify column-major for `addMatrix` by using `coder.columnMajor`.

```
function S = addMatrix(A,B)
%#codegen
S = zeros(size(A));
coder.columnMajor; % specify column-major array layout
S = A + B;
end
```

Generate code for `myFunction`. Use the `codegen` command.

```
codegen myFunction -args {ones(10,20),ones(10,20)} -config:lib -launchreport -rowmajor
```

Because of the `codegen -rowmajor` option, the matrix multiplication in `myFunction` uses row-major layout. However, the generated code for `addMatrix` uses column-major array layout due to the `coder.columnMajor` call.

Tips

- The code generator uses column-major array layout by default.
- The specification of array layout inside a function supersedes the array layout specified with the `codegen` command. For example, if the function `foo` contains `coder.columnMajor`, and you generate code by using:

  ```
  codegen foo -rowmajor
  ```

  then the generated code still uses column-major layout.
- Other functions called from within a column-major function inherit the column-major specification. However, if one of the called functions has its own distinct `coder.rowMajor` call, the code generator changes the array layout accordingly. If a row-major function and a column-major function call the same function, which does not have its own array layout specification, the code generator produces a row-major version and column-major version of the function.
- `coder.columnMajor` is ignored outside of code generation.

See Also

`coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

Topics

“Row-Major and Column-Major Array Layouts”
“Generate Code That Uses Row-Major Array Layout”
“Specify Array Layout in Functions and Classes”
“Generate Code That Uses N-Dimensional Indexing”

Introduced in R2018a

coder.config

Package: coder

Create MATLAB Coder code generation configuration objects

Syntax

```
config_obj = coder.config
config_obj = coder.config(build_type)
config_obj = coder.config(build_type, 'ecoder', ecoder_flag)
config_obj = coder.config(numeric_conversion_type)
```

Description

`config_obj = coder.config` creates a `coder.MexCodeConfig` code generation configuration object for use with `codegen` when generating a MEX function. Use a `coder.MexCodeConfig` object with the `-config` option of the `codegen` command.

`config_obj = coder.config(build_type)` creates a code generation configuration object for use with `codegen` when generating a MEX function or standalone code (static library, dynamically linked library or executable program). Use the code generation configuration object with the `-config` option of the `codegen` command.

`config_obj = coder.config(build_type, 'ecoder', ecoder_flag)` creates a `coder.EmbeddedCodeConfig` object or a `coder.CodeConfig` object depending on whether `ecoder_flag` is `true` or `false`. `build_type` is `'lib'`, `'dll'`, or `'exe'`.

`config_obj = coder.config(numeric_conversion_type)` creates these configuration objects for use with `codegen`:

- `coder.FixptConfig` when generating fixed-point MATLAB or C/C++ code from floating-point MATLAB code. Use with the `-float2fixed` option of the `codegen` command.
- `coder.SingleConfig` when generating single-precision MATLAB code from double-precision MATLAB code. Use with the `-double2single` option of the `codegen` command.

Fixed-point conversion or single-precision conversion requires Fixed-Point Designer.

Examples

Generate MEX Function from MATLAB Function

Generate a MEX function from a MATLAB function that is suitable for code generation and enable a code generation report.

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
% The directive %#codegen declares that the function
% is intended for code generation
r = rand();
```

2. Create a code generation configuration object to generate a MEX function.

```
cfg = coder.config % or cfg = coder.config('mex')
```

3. Open the code generation report.

```
cfg.GenerateReport = true;
```

4. Generate a MEX function in the current folder that specifies the configuration object by using the `-config` option.

```
% Generate a MEX function and code generation report
codegen -config cfg coderand
```

Generate Standalone C Static or Dynamic Library or Generate Standalone C Executable

Create a code generation configuration object for a standalone C static library.

```
cfg = coder.config('lib')
% Returns a coder.EmbeddedCodeConfig object if the Embedded
% Coder product is installed.
% Otherwise, returns a coder.CodeConfig object.
```

Create a code generation configuration object to generate a standalone C dynamic library.

```
cfg = coder.config('dll')
% Returns a coder.EmbeddedCodeConfig object if the Embedded
% Coder product is installed.
% Otherwise, returns a coder.CodeConfig object.
```

Create a code generation configuration object to generate a standalone C executable.

```
cfg = coder.config('exe')
% Returns a coder.EmbeddedCodeConfig object if the Embedded
% Coder product is installed.
% Otherwise, returns a coder.CodeConfig object.
```

Create a `coder.CodeConfig` Object with Embedded Coder Installed

Create a `coder.CodeConfig` object even when the Embedded Coder product is installed on your system.

```
cfg = coder.config('lib','ecoder',false)
```

Create a `coder.EmbeddedCodeConfig` object without Embedded Coder.

```
cfg = coder.config('lib','ecoder',true)
```

Create a Numeric Conversion Configuration Object

Create a `coder.FixptConfig` object.

```
fixptcfg = coder.config('fixpt');
```

Create a `coder.SingleConfig` object.

```
scfg = coder.config('single');
```

Input Arguments

`build_type` — Type of code generation object to create

'mex' | 'lib' | 'dll' | 'exe'

Configuration Object Type	Generated Code	Code Generation Configuration Object (Embedded Coder installed)	Code Generation Configuration Object (Embedded Coder not installed)
`'mex'`	MEX function	`coder.MexCodeConfig`	`coder.MexCodeConfig`
`'lib'`	Static library	`coder.EmbeddedCodeConfig`	`coder.CodeConfig`
`'dll'`	Dynamic library	`coder.EmbeddedCodeConfig`	`coder.CodeConfig`
`'exe'`	Executable	`coder.EmbeddedCodeConfig`	`coder.CodeConfig`

Example: coder.config('mex');

Data Types: `char` | `string`

`numeric_conversion_type` — Numeric conversion object type
'fixpt' | 'single'

`'fixpt'`	Creates a `coder.FixptConfig` configuration object for use with `codegen` when generating fixed-point MATLAB or C/C++ code from floating-point MATLAB code.
`'single'`	Creates a `coder.SingleConfig` configuration object for use with `codegen` when generating single-precision MATLAB code from double-precision MATLAB code.

Example: coder.config('fixpt');

Data Types: `char` | `string`

`ecoder_flag` — Embedded Coder code configuration object flag
false | true

`true`	Creates a `coder.EmbeddedCodeConfig` configuration object without Embedded Coder. However, code generation by using a `coder.EmbeddedCodeConfig` object requires the Embedded Coder product. `build_type` must be `'lib'`, `'dll'`, or `'exe'`.
`false`	Creates a `coder.CodeConfig` configuration object even if the Embedded Coder product is installed. `build_type` must be `'lib'`, `'dll'`, or `'exe'`.

Example: coder.config('lib', 'ecoder', false);

Data Types: `logical`

Output Arguments

`config_obj` — Code generation configuration handle
coder.CodeConfig | coder.MexCodeConfig | coder.EmbeddedCodeConfig | coder.FixptConfig | coder.SingleConfig

Handle to the MATLAB Coder code generation configuration object.

Alternatives

Use the `coder` function to open the MATLAB Coder app and create a MATLAB Coder project. The app provides a user interface that facilitates adding MATLAB files, defining input parameters, and specifying build parameters.

See Also

`codegen` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.FixptConfig` | `coder.MexCodeConfig` | `coder.SingleConfig`

Topics

“Accelerate MATLAB Algorithm by Generating MEX Function”
“Generate C Code at the Command Line”
“Convert MATLAB Code to Fixed-Point C Code”
“Generate Single-Precision C Code at the Command Line”

Introduced in R2011a

coder.const

Fold expressions into constants in generated code

Syntax

```
out = coder.const(expression)
[out1,...,outN] = coder.const(handle,arg1,...,argN)
```

Description

`out = coder.const(expression)` evaluates `expression` and replaces `out` with the result of the evaluation in generated code.

`[out1,...,outN] = coder.const(handle,arg1,...,argN)` evaluates the multi-output function having handle `handle`. It then replaces `out1,...,outN` with the results of the evaluation in the generated code.

Examples

Specify Constants in Generated Code

This example shows how to specify constants in generated code using `coder.const`.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator produces code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int k;
  for (k = 0; k < 10; k++) {
    y[k] = (double)((1 + k) * (1 + k)) + Shift;
  }
}
```

Replace the expression `(1:10).^2` with `coder.const((1:10).^2)`, and then generate code for `AddShift` again using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int i0;
  static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                       49, 64, 81, 100 };

  for (i0 = 0; i0 < 10; i0++) {
    y[i0] = (double)iv0[i0] + Shift;
  }
}
```

Create Lookup Table in Generated Code

This example shows how to fold a user-written function into a constant in generated code.

Write a function `getsine` that takes an input `index` and returns the element referred to by `index` from a lookup table of sines. The function `getsine` creates the lookup table using another function `gettable`.

```
function y = getsine(index) %#codegen
  assert(isa(index, 'int32'));
```

```
  persistent tbl;
  if isempty(tbl)
           tbl = gettable(1024);
  end
  y = tbl(index);

function y = gettable(n)
      y = zeros(1,n);
      for i = 1:n
         y(i) = sin((i-1)/(2*pi*n));
      end
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

```
codegen -config:lib -launchreport getsine -args int32(0)
```

The generated code contains instructions for creating the lookup table.

Replace the statement:

```
tbl = gettable(1024);
```

with:

```
tbl = coder.const(gettable(1024));
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

The generated code contains the lookup table itself. `coder.const` forces the expression `gettable(1024)` to be evaluated during code generation. The generated code does not contain instructions for the evaluation. The generated code contains the result of the evaluation itself.

Specify Constants in Generated Code Using Multi-Output Function

This example shows how to specify constants in generated code using a multi-output function in a `coder.const` statement.

Write a function `MultiplyConst` that takes an input `factor` and multiplies every element of two vectors `vec1` and `vec2` with `factor`. The function generates `vec1` and `vec2` using another function `EvalConsts`.

```
function [y1,y2] = MultiplyConst(factor) %#codegen
  [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
  y1=vec1.*factor;
  y2=vec2.*factor;

function [f1,f2]=EvalConsts(z,n)
  f1=z.^(2*n)/factorial(2*n);
  f2=z.^(2*n+1)/factorial(2*n+1);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator produces code for creating the vectors.

Replace the statement

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

with

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator does not generate code for creating the vectors. Instead, it calculates the vectors and specifies the calculated vectors in generated code.

Read Constants by Processing XML File

This example shows how to call an extrinsic function using `coder.const`.

Write an XML file `MyParams.xml` containing the following statements:

```
<params>
    <param name="hello" value="17"/>
```

```
    <param name="world" value="42"/>
</params>
```

Save `MyParams.xml` in the current folder.

Write a MATLAB function `xml2struct` that reads an XML file. The function identifies the XML tag `param` inside another tag `params`.

After identifying `param`, the function assigns the value of its attribute `name` to the field name of a structure `s`. The function also assigns the value of attribute `value` to the value of the field.

```
function s = xml2struct(file)

s = struct();
doc = xmlread(file);
els = doc.getElementsByTagName('params');
for i = 0:els.getLength-1
    it = els.item(i);
    ps = it.getElementsByTagName('param');
    for j = 0:ps.getLength-1
        param = ps.item(j);
        paramName = char(param.getAttribute('name'));
        paramValue = char(param.getAttribute('value'));
        paramValue = evalin('base', paramValue);
        s.(paramName) = paramValue;
    end
end
```

Save `xml2struct` in the current folder.

Write a MATLAB function `MyFunc` that reads the XML file `MyParams.xml` into a structure `s` using the function `xml2struct`. Declare `xml2struct` as extrinsic using `coder.extrinsic` and call it in a `coder.const` statement.

```
function y = MyFunc(u) %#codegen
  assert(isa(u, 'double'));
  coder.extrinsic('xml2struct');
  s = coder.const(xml2struct('MyParams.xml'));
  y = s.hello + s.world + u;
```

Generate code for `MyFunc` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:dll -launchreport MyFunc -args 0
```

The code generator executes the call to `xml2struct` during code generation. It replaces the structure fields `s.hello` and `s.world` with the values 17 and 42 in generated code.

Input Arguments

`expression` — MATLAB expression or user-written function

expression with constants | single-output function with constant arguments

MATLAB expression or user-defined single-output function.

The expression must have compile-time constants only. The function must take constant arguments only. For instance, the following code leads to a code generation error, because `x` is not a compile-time constant.

```
function y=func(x)
   y=coder.const(log10(x));
```

To fix the error, assign `x` to a constant in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args coder.Constant(10)
```

Example: `2*pi`, `factorial(10)`

`handle` — Function handle

function handle

Handle to built-in or user-written function.

Example: `@log`, `@sin`

Data Types: `function_handle`

`arg1,...,argN` — Arguments to the function with handle `handle`

function arguments that are constants

Arguments to the function with handle `handle`.

The arguments must be compile-time constants. For instance, the following code leads to a code generation error, because `x` and `y` are not compile-time constants.

```
function y=func(x,y)
   y=coder.const(@nchoosek,x,y);
```

To fix the error, assign `x` and `y` to constants in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

Output Arguments

`out` — Value of `expression`
value of the evaluated expression

Value of `expression`. In the generated code, MATLAB Coder replaces occurrences of `out` with the value of `expression`.

`out1,...,outN` — Outputs of the function with handle `handle`
values of the outputs of the function with handle `handle`

Outputs of the function with handle `handle`.MATLAB Coder evaluates the function and replaces occurrences of `out1,...,outN` with constants in the generated code.

Tips

- When possible, the code generator constant-folds expressions automatically. Typically, automatic constant-folding occurs for expressions with scalars only. Use `coder.const` when the code generator does not constant-fold expressions on its own.
- When constant-folding computationally intensive function calls, to reduce code generation time, make the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

  ```
  function j = fcn(z)
  zTable = coder.const(0:0.01:100);
  jTable = coder.const(feval('besselj',3,zTable));
  j = interp1(zTable,jTable,z);
  end
  ```

  See “Use coder.const with Extrinsic Function Calls”.
- If `coder.const` is unable to constant-fold a function call, try to force constant-folding by making the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

```
function yi = fcn(xi)
y = coder.const(feval('rand',1,100));
yi = interp1(y,xi);
end
```

See “Use coder.const with Extrinsic Function Calls”.

See Also

Topics

“Constant Folding”
“Fold Function Calls into Constants”
“Use coder.const with Extrinsic Function Calls”

Introduced in R2013b

coder.cstructname

Package: coder

Name C structure type in generated code

`coder.cstructname` names the generated or externally defined C structure type to use for MATLAB variables that are represented as structures in generated code.

Syntax

```
coder.cstructname(var,structName)
coder.cstructname(var,structName,'extern','HeaderFile',headerfile)
coder.cstructname(var,structName,'extern','HeaderFile',
headerfile,'Alignment',alignment)

outtype = coder.cstructname(intype,structName)
outtype = coder.cstructname(intype,structName,'extern','HeaderFile',
headerfile)
outtype = coder.cstructname(intype,structName,'extern','HeaderFile',
headerfile,'Alignment',alignment)
```

Description

`coder.cstructname(var,structName)` names the C structure type generated for the MATLAB variable `var`. The input `var` can be a structure or a cell array. Use this syntax in a function from which you generate code. Place `coder.cstructname` after the definition of `var` and before the first use of `var`. If `var` is an entry-point (top-level) function input argument, place `coder.cstructname` at the beginning of the function, before any control flow statements.

`coder.cstructname(var,structName,'extern','HeaderFile',headerfile)` specifies that the C structure type to use for `var` has the name `structName` and is defined in the external file, `headerfileName`.

It is possible to use the `'extern'` option without specifying the header file. However, it is a best practice to specify the header file so that the code generator produces the `#include` statement in the correct location.

`coder.cstructname(var,structName,'extern','HeaderFile', headerfile,'Alignment',alignment)` also specifies the run-time memory alignment for the externally defined structure type `structName`. If you have Embedded Coder and use custom Code Replacement Libraries (CRLs), specify the alignment so that the code generator can match CRL functions that require alignment for structures. See “Data Alignment for Code Replacement” (Embedded Coder).

`outtype = coder.cstructname(intype,structName)` returns a structure or cell array type object `outtype` that specifies the name of the C structure type to generate. `coder.cstructname` creates `outtype` with the properties of the input type `intype`. Then, it sets the `TypeName` property to `structName`. Use this syntax to create a type object that you use with the `codegen -args` option. You cannot use this syntax in a function from which you generate code.

You cannot use this syntax in a MATLAB Function block.

`outtype = coder.cstructname(intype,structName,'extern','HeaderFile', headerfile)` returns a type object `outtype` that specifies the name and location of an externally defined C structure type. The code generator uses the externally defined structure type for variables with type `outtype`.

You cannot use this syntax in a MATLAB Function block.

`outtype = coder.cstructname(intype,structName,'extern','HeaderFile', headerfile,'Alignment',alignment)` creates a type object `outtype` that also specifies the C structure type alignment.

You cannot use this syntax in a MATLAB Function block.

Examples

Name the C Structure Type for a Variable in a Function

In a MATLAB function, `myfun`, assign the name `MyStruct` to the generated C structure type for the variable `v`.

```
function y = myfun()
%#codegen
v = struct('a',1,'b',2);
coder.cstructname(v, 'myStruct');
y = v;
end
```

Generate standalone C code. For example, generate a static library.

```
codegen -config:lib myfun -report
```

To see the generated structure type, open `codegen/lib/myfun/myfun_types.h` or view `myfun_types.h` in the code generation report. The generated C structure type is:

```
typedef struct {
  double a;
  double b;
} myStruct;
```

Name the C Structure Type Generated for a Substructure

In a MATLAB function, `myfun1`, assign the name `MyStruct` to the generated C structure type for the structure `v`. Assign the name `mysubStruct` to the structure type generated for the substructure `v.b`.

```
function y = myfun()
%#codegen
v = struct('a',1,'b',struct('f',3));
coder.cstructname(v, 'myStruct');
coder.cstructname(v.b, 'mysubStruct');
y = v;
end
```

The generated C structure type `mysubStruct` is:

```
typedef struct {
  double f;
} mysubStruct;
```

The generated C structure type `myStruct` is:

```
typedef struct {
  double a;
```

```
  mysubStruct b;
} myStruct;
```

Name the Structure Type Generated for a Cell Array

In a MATLAB function, `myfun2`, assign the name `myStruct` to the generated C structure type for the cell array`c`.

```
function z = myfun2()
c = {1 2 3};
coder.cstructname(c,'myStruct')
z = c;
```

The generated C structure type for `c` is:

```
typedef struct {
  double f1;
  double f2;
  double f3;
} myStruct;
```

Name an Externally Defined C Structure Type

Specify that a structure passed to a C function has a structure type defined in a C header file.

Create a C header file `mycadd.h` for the function `mycadd` that takes a parameter of type `mycstruct`. Define the type `mycstruct` in the header file.

```
#ifndef MYCADD_H
#define MYCADD_H

typedef struct {
    double f1;
    double f2;
} mycstruct;

double mycadd(mycstruct *s);
#endif
```

Write the C function `mycadd.c`.

```
#include <stdio.h>
#include <stdlib.h>

#include "mycadd.h"

double mycadd(mycstruct *s)
{
  return  s->f1 + s->f2;
}
```

Write a MATLAB function `mymAdd` that passes a structure by reference to `mycadd`. Use `coder.cstructname` to specify that in the generated code, the structure has the C type `mycstruct`, which is defined in `mycadd.h`.

```
function y = mymAdd
%#codegen
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'mycstruct', 'extern', 'HeaderFile', 'mycadd.h');
y = 0;
y = coder.ceval('mycadd', coder.ref(s));
```

Generate a C static library for function `mymAdd`.

```
codegen -config:lib mymAdd mycadd.c
```

The generated header file `mymadd_types.h` does not contain a definition of the structure `mycstruct` because `mycstruct` is an external type.

Create a Structure Type Object That Names the Generated C Structure Type

Suppose that the entry-point function `myFunction` takes a structure argument. To specify the type of the input argument at the command line:

1. Define an example structure `S`.
2. Create a type `T` from `S` by using `coder.typeof`.
3. Use `coder.cstructname` to create a type `T1` that:
   - Has the properties of `T`.
   - Names the generated C structure type `myStruct`.
4. Pass the type to `codegen` by using the `-args` option.

For example:

```
S = struct('a',double(0),'b',single(0));
T = coder.typeof(S);
T1 = coder.cstructname(T,'myStruct');
codegen -config:lib myFunction -args T1
```

Alternatively, you can create the structure type directly from the example structure.

```
S = struct('a',double(0),'b',single(0));
T1 = coder.cstructname(S,'myStruct');
codegen -config:lib myFunction -args T1
```

Input Arguments

`var` — MATLAB structure or cell array variable

structure | cell array

MATLAB structure or cell array variable that is represented as a structure in the generated code.

`structName` — Name of C structure type

character vector | string scalar

Name of generated or externally defined C structure type, specified as a character vector or string scalar.

`headerfile` — Header file that contains the C structure type definition

character vector | string scalar

Header file that contains the C structure type definition, specified as a character vector or string scalar.

To specify the path to the file:

- Use the `codegen -I` option or the **Additional include directories** parameter on the MATLAB Coder app settings **Custom Code** tab.
- For a MATLAB Function block, on the **Simulation Target** and the **Code Generation > Custom Code** panes, under **Additional build information**, set the **Include directories** parameter.

Alternatively, use `coder.updateBuildInfo` with the `'addIncludePaths'` option.

Example: `'mystruct.h'`

`alignment` — Run-time memory alignment for structure
-1 (default) | power of 2 not greater than 128

Run-time memory alignment for generated or externally defined structure.

`intype` — Type object or variable for creation of new type object
`coder.StructType` | `coder.CellType` | structure | cell array

Structure type object, cell array type object, structure variable, or cell array variable from which to create a type object.

Limitations

- You cannot apply `coder.cstructname` directly to a global variable. To name the structure type to use with a global variable, use `coder.cstructname` to create a type object that names the structure type. Then, when you run `codegen`, specify that the global variable has that type. See “Name the C Structure Type to Use With a Global Structure Variable”.
- For cell array inputs, the field names of externally defined structures must be `f1`, `f2`, and so on.

Tips

- For information about how the code generator determines the C/C++ types of structure fields, see “Mapping MATLAB Types to Types in Generated Code”.
- Using `coder.cstructname` on a structure array sets the name of the structure type of the base element, not the name of the array. Therefore, you cannot apply `coder.cstructname` to a structure array element, and then apply it to the array with a different C structure type name. For example, the following code is not allowed. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previously specified name, `myStructName`.

```
% Define scalar structure with field a
myStruct = struct('a', 0);
coder.cstructname(myStruct,'myStructName');
```

```
% Define array of structure with field a
myStructArray = repmat(myStruct,k,n);
coder.cstructname(myStructArray,'myStructArrayName');
```

- Applying `coder.cstructname` to an element of a structure array produces the same result as applying `coder.cstructname` to the entire structure array. If you apply `coder.cstructname` to an element of a structure array, you must refer to the element by using a single subscript. For example, you can use `var(1)`, but not `var(1,1)`. Applying `coder.cstructname` to `var(:)` produces the same result as applying `coder.cstructname` to `var` or `var(n)`.
- Heterogeneous cell arrays are represented as structures in the generated code. Here are considerations for using `coder.cstructname` with cell arrays:
  - In a function from which you generate code, using `coder.cstructname` with a cell array variable makes the cell array heterogeneous. Therefore, if a cell array is an entry-point function input and its type is permanently homogeneous, then you cannot use `coder.cstructname` with the cell array.
  - Using `coder.cstructname` with a homogeneous `coder.CellType` object `intype` makes the returned object heterogeneous. Therefore, you cannot use `coder.cstructname` with a permanently homogeneous `coder.CellType` object. For information about when a cell array is permanently homogeneous, see “Specify Cell Array Inputs at the Command Line”.
  - When used with a `coder.CellType` object, `coder.cstructname` creates a `coder.CellType` object that is permanently heterogeneous.
- When you use a structure named by `coder.cstructname` in a project with row-major and column-major array layouts, the code generator renames the structure in certain cases, appending `row_` or `col_` to the beginning of the structure name. This renaming provides unique type definitions for the types that are used in both array layouts.
- These tips apply only to MATLAB Function blocks:
  - MATLAB Function block input and output structures are associated with bus signals. The generated name for the structure type comes from the bus signal name. Do not use `coder.cstructname` to name the structure type for input or output signals. See “Create Structures in MATLAB Function Blocks” (Simulink).
  - The code generator produces structure type names according to identifier naming rules, even if you name the structure type with `coder.cstructname`. If you have Embedded Coder, you can customize the naming rules. See “Construction of Generated Identifiers” (Embedded Coder).

See Also

`codegen` | `coder.CellType` | `coder.StructType` | `coder.ceval`

Topics

“Structure Definition for Code Generation”
“Code Generation for Cell Arrays”
“Specify Cell Array Inputs at the Command Line”
“Call C/C++ Code from MATLAB Code”

Introduced in R2011a

coder.DeepLearningConfig

Create deep learning code generation configuration objects

Syntax

```
deepLearningCfg = coder.DeepLearningConfig(targetlib)
```

Description

`deepLearningCfg = coder.DeepLearningConfig(targetlib)` creates a deep learning configuration object containing library-specific parameters that `codegen` uses to generate code for deep neural networks. Assign this deep learning configuration object to the `DeepLearningConfig` property of the code configuration object created by using `coder.config`. Pass the code configuration object to the `codegen` function by using the `-config` option.

Examples

Generate Code for the AlexNet Network Using Intel MKL-DNN Library

Set the code configuration parameters and generate C++ code for an `AlexNet` series network. The generated code uses the Intel MKL-DNN deep learning libraries.

Create an entry-point function `alexneteg` that uses the `coder.loadDeepLearningNetwork` function to load the `alexnet` `SeriesNetwork` object.

```
function out = alexneteg(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('alexnet', 'myalexnet');
end
```

```
out = predict(mynet,in);
```

The persistent object avoids reconstructing and reloading the network object during subsequent calls to the function to invoke the `predict` method on the input

The input layer of the pretrained `AlexNet` network accepts images of size `227x227x3`. To read an input image from a graphics file and resize it to `227x227`, use the following lines of code:

```
in = imread('peppers.png');
in = imresize(in,[227,227]);
```

Create a `coder.config` configuration object for MEX code generation and set the target language to C++. On the configuration object, set `DeepLearningConfig` with `targetlib` as `'mkldnn'`. Use the `-config` option of the `codegen` function to pass this code configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -args {ones(227,227,3,'single')} -config cfg alexneteg
```

The `codegen` command places all the generated files in the `codegen` folder. It contains the C++ code for the entry-point function `alexneteg.cpp`, header and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Input Arguments

`targetlib` — Specify the target deep learning library

character vector | string scalar

Target library for deep learning code generation, specified as one of the values in this table.

Value	Description
`'arm-compute'`	For generating code that uses the ARM Compute Library.
`'mkldnn'`	For generating code that uses theIntel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).
`'cudnn'`	For generating code that uses the CUDA Deep Neural Network library (cuDNN). This option requires GPU Coder.
`'tensorrt'`	For generating code that takes advantage of the NVIDIA TensorRT – high performance deep learning inference optimizer and run-time library. This option requires GPU Coder.

Output Arguments

`deepLearningCfg` — Deep learning configuration object
`Configuration Object`

Configuration object based on the target library specified in the input argument. This object contains library-specific parameters that are used during code generation.

Target Library	Deep Learning Configuration Object
`'arm-compute'`	Creates an `ARMNEONConfig` configuration object.
`'mkldnn'`	Creates an `MklDNNConfig` configuration object.
`'cudnn'`	Creates a `CuDNNConfig` configuration object.
`'tensorrt'`	Creates a `TensorRTConfig` configuration object.

See Also

`codegen` | `coder.ARMNEONConfig` | `coder.CodeConfig` | `coder.MklDNNConfig` | `coder.loadDeepLearningNetwork`

Topics

“Code Generation for Deep Learning Networks with MKL-DNN”
“Code Generation for Deep Learning Networks with ARM Compute Library”
“Code Generation for Object Detection Using YOLO v2” (GPU Coder)
“Code Generation for Deep Learning Networks with cuDNN” (GPU Coder)
“Code Generation for Deep Learning Networks with TensorRT” (GPU Coder)

Introduced in R2018b

coder.extrinsic

Package: coder

Declare extrinsic function or functions

Syntax

```
coder.extrinsic('function_name');

coder.extrinsic('function_name_1', ... , 'function_name_n');

coder.extrinsic('-sync:on', 'function_name');

coder.extrinsic('-sync:on', 'function_name_1', ... ,
'function_name_n');

coder.extrinsic('-sync:off','function_name');

coder.extrinsic('-sync:off', 'function_name_1', ... ,
'function_name_n');
```

Arguments

function_name
function_name_1, ... , *function_name_n*

Declares *function_name* or *function_name_1* through *function_name_n* as extrinsic functions.

–sync:on

function_name or *function_name_1* through *function_name_n*.

Enables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *function_name* or *function_name_1* through *function_name_n*. If only a few extrinsic calls modify global data, turn off synchronization before and after all extrinsic function calls by setting the global synchronization mode to `At MEX-function entry and exit`. Use the *–sync:on*

option to turn on synchronization for only the extrinsic calls that *do* modify global data.

For constant global data, enables verification of consistency between MATLAB and MEX functions after calls to the extrinsic functions, *`function_name`* or *`function_name_1`* through *`function_name_n`*.

`-sync:off`

Disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *`function_name`* or *`function_name_1`* through *`function_name_n`*. If most extrinsic calls modify global data, but a few do not, you can use the *-sync:off* option to turn off synchronization for the extrinsic calls that *do not* modify global data.

For constant global data, disables verification of consistency between MATLAB and MEX functions after calls to the extrinsic functions, *`function_name`* or *`function_name_1`* through *`function_name_n`*.

Description

`coder.extrinsic` declares extrinsic functions. During simulation, the code generator produces code for the call to an extrinsic function, but does not produce the function's internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that there is no change to the output, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

You cannot use `coder.ceval` on functions that you declare extrinsic by using `coder.extrinsic`.

`coder.extrinsic` is ignored outside of code generation.

Limitations

- Extrinsic function calls have some overhead that can affect performance. Input data that is passed in an extrinsic function call must be provided to MATLAB, which

requires making a copy of the data. If the function has any output data, this data must be transferred back into the MEX function environment, which also requires a copy.

- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.

Tips

- The code generator detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions, but you do not have to declare them extrinsic using the `coder.extrinsic` function.
- Use the `coder.screener` function to detect which functions you must declare extrinsic. This function opens the code generations readiness tool that detects code generation issues in your MATLAB code.

Examples

The following code declares the MATLAB function `patch` as extrinsic in the MATLAB local function `create_plot`.

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
%  and displays the triangle as a patch object.

c = sqrt(a^2 + b^2);

create_plot(a, b, color);

function create_plot(a, b, color)

%Declare patch as extrinsic
coder.extrinsic('patch');

x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```

By declaring `patch` as extrinsic, you instruct the code generator not to compile or produce code for `patch`. Instead, the code generator dispatches `patch` to MATLAB for execution.

See Also

`coder.ceval` | `coder.screener`

Topics

“Extrinsic Functions”
“Controlling Synchronization for Extrinsic Function Calls”
“Define Constant Global Data”
“Resolution of Function Calls for Code Generation”
“Restrictions on Extrinsic Functions for Code Generation”

Introduced in R2011a

coder.getArgTypes

Determine types of function input arguments by running test file

Syntax

```
types = coder.getArgTypes(test_fcn,fcn)
structure_of_types = coder.getArgTypes(test_fcn, {fcn_1,...,fcn_n})
structure_of_types = coder.getArgTypes(test_fcn,fcn,'uniform',true)
```

Description

`types = coder.getArgTypes(test_fcn,fcn)` returns a cell array of `coder.Type` objects determined by executing `test_fcn`. `test_fcn` should call the specified entry-point MATLAB function, `fcn`. The software uses the input arguments to `fcn` to construct the returned types.

`structure_of_types = coder.getArgTypes(test_fcn, {fcn_1,...,fcn_n})` returns a structure containing cell arrays of `coder.Type` objects determined by executing `test_fcn`. `test_fcn` should call the specified entry-point functions, `fcn_1` through `fcn_n`. The software uses the input arguments to these functions to construct the returned types. The returned structure contains one field for each function. The field name is the same as the name of the corresponding function.

`structure_of_types = coder.getArgTypes(test_fcn,fcn,'uniform',true)` returns a structure even though there is only one entry-point function.

Input Arguments

`fcn`

Name or handle of entry-point MATLAB function for which you want to determine input types. The function must be on the MATLAB path; it cannot be a local function. The function must be in a writable folder.

`fcn_1,...,fcn_n`

Comma-separated list of names or handles of entry-point MATLAB functions for which you want to determine input types. The functions must be on the MATLAB path; they cannot be a local function. The functions must be in a writable folder. The entry-point function names must be unique.

`test_fcn`

Name or handle of test function or name of test script. The test function or script must be on the MATLAB path. `test_fcn` should call at least one of the specified entry-point functions. The software uses the input arguments to these functions to construct the returned types.

Output Arguments

`types`

Cell array of `coder.Type` objects determined by executing the test function.

`structure_of_types`

Structure containing cell arrays of `coder.Type` objects determined by executing the `test_fcn`. The structure contains one field for each function. The field name is the same as the name of the corresponding function.

Examples

Get input parameter types for one entry-point function

Get input parameter types for function `my_fun` by running test file `my_test` that calls `my_fun`. Use these input types to generate code for `my_fun`.

In a local writable folder, create the MATLAB function.

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

In the same folder, create the test function.

```
function y = my_test
    a = single(10);
     b = single(20);
    y = my_fun(a,b);
end
```

Run the test function to get the input types for `my_fun`.

```
types = coder.getArgTypes('my_test','my_fun')
```

```
types =

    [1x1 coder.PrimitiveType]    [1x1 coder.PrimitiveType]
```

Generate a MEX function for `my_fun` using these input types as example inputs.

```
codegen my_fun -args types
```

In the current folder, `codegen` generates a MEX function, `my_fun_mex`, that accepts inputs of type `single`.

You can now test the MEX function. For example:

```
 y = my_fun_mex(single(11),single(22))
```

Get input types for multiple entry-point functions

Get input parameter types for functions `my_fun1` and `my_fun2` by running test file `my_test2` that calls `my_fun1` and `my_fun2`. Use these input types to generate code for `my_fun1` and `my_fun2`.

In a local writable folder, create the MATLAB function, `my_fun1`.

```
function y = my_fun1(u) %#codegen
y = u;
```

In the same folder, create the function, `my_fun2`.

```
function y = my_fun2(u, v) %#codegen
y = u + v;
```

In the same folder, create the test function.

```
function [y1, y2] =  my_test2
    a = 10;
    b = 20;
    y1 = my_fun1(a);
    y2 = my_fun2(a,b);
end
```

Run the test function to get the input types for `my_fun1` and `my_fun2`.

```
types = coder.getArgTypes('my_test2',{'my_fun1','my_fun2'})
```

```
types =

    my_fun1: {[1x1 coder.PrimitiveType]}
    my_fun2: {[1x1 coder.PrimitiveType]  [1x1 coder.PrimitiveType]}
```

Generate a MEX function for `my_fun1` and `my_fun2` using these input types as example inputs.

```
codegen my_fun1 -args types.my_fun1 my_fun2 -args types.my_fun2
```

In the current folder, `codegen` generates a MEX function, `my_fun1_mex`, with two entry points, `my_fun1` and `my_fun2`, that accept inputs of type `double`.

You can now test each entry point in the MEX function. For example:

```
y1 = my_fun1_mex('my_fun1',10)
y2 = my_fun1_mex('my_fun2',15, 25)
```

Tips

- Before using `coder.getArgTypes`, run the test function in MATLAB to verify that it provides the expected results.
- Verify that the test function calls the specified entry-point functions with input data types suitable for your runtime environment. If the test function does not call a specified function, `coder.getArgTypes` cannot determine the input types for this function.
- `coder.getArgTypes` might not compute the ideal type for your application. For example, you might want the size to be unbounded. `coder.getArgTypes` returns a bound based on the largest input that it has seen. Use `coder.resize` to adjust the sizes of the returned types.

- For some combinations of inputs, `coder.getArgTypes` cannot produce a valid type. For example, if the test function calls the entry-point function with single inputs and then calls it with double inputs, `coder.getArgTypes` generates an error because there is no single type that can represent both calls.
- When you generate code for the MATLAB function, use the returned types as example inputs by passing them to the `codegen` using the `-args` option.

Alternatives

- “Specify Properties of Entry-Point Function Inputs Using the App”
- “Define Input Properties Programmatically in the MATLAB File”

See Also

`codegen` | `coder.resize` | `coder.runTest` | `coder.typeof`

Topics

“Specify Properties of Entry-Point Function Inputs”

Introduced in R2012a

coder.getDeepLearningLayers

Get convolutional neural network layers supported for code generation for a specific deep learning library

Syntax

```
coder.getDeepLearningLayers(libraryname)
```

Description

`coder.getDeepLearningLayers(libraryname)` returns the convolutional neural network layers supported for code generation for a specific deep learning library.

Note To use `coder.getDeepLearningLayers`, you must install the support package that corresponds to `libraryname`:

- For `'arm-compute'` and `'mkldnn'`, install MATLAB Coder Interface for Deep Learning Libraries.
- For `'cudnn'` and `'tensorrt'`, install GPU Coder Interface for Deep Learning Libraries.

Examples

Get Layers Supported for Code Generation for a Specific Deep Learning Library

Get a list of layers supported for code generation for Intel Math Kernel Library for Deep Neural Networks.

```
coder.getDeepLearningLayers('mkldnn')
```

```
ans =
```

```
17×1 cell array

  {'AdditionLayer'                 }
  {'AveragePooling2DLayer'         }
  {'BatchNormalizationLayer'       }
  {'ClassificationOutputLayer'     }
  {'ClippedReLULayer'              }
  {'Convolution2DLayer'            }
  {'CrossChannelNormalizationLayer'}
  {'DepthConcatenationLayer'       }
  {'DropoutLayer'                  }
  {'FullyConnectedLayer'           }
  {'ImageInputLayer'               }
  {'LeakyReLULayer'                }
  {'MaxPooling2DLayer'             }
  {'ReLULayer'                     }
  {'RegressionOutputLayer'         }
  {'SoftmaxLayer'                  }
  {'TransposedConvolution2DLayer'  }
```

Input Arguments

`libraryname` — Name of deep learning library

character vector | string scalar

Name of deep learning library, specified as one of the values in this table.

Value	Description
`'arm-compute'`	ARM Compute Library Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'cudnn'`	cuDNN library Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.

Value	Description
`'mkldnn'`	Intel Math Kernel Library for Deep Neural Networks Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'tensorrt'`	TensorRT library Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.

See Also

Topics

“Installing Prerequisite Products” (GPU Coder)
“Prerequisites for Deep Learning with MATLAB Coder”
“Supported Networks and Layers” (GPU Coder)
“Deep Learning Networks and Layers Supported for C++ Code Generation”
“Code Generation for Deep Learning Networks with MKL-DNN”
“Code Generation for Deep Learning Networks with ARM Compute Library”
“Code Generation for Deep Learning Networks with cuDNN” (GPU Coder)
“Code Generation for Deep Learning Networks with TensorRT” (GPU Coder)

Introduced in R2018b

coder.ignoreConst

Prevent use of constant value of expression for function specializations

Syntax

```
coder.ignoreConst(expression)
```

Description

`coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations on page 2-104. `coder.ignoreConst(expression)` returns the value of `expression`.

Examples

Prevent Function Specializations Based on Constant Input Values

Use `coder.ignoreConst` to prevent function specializations for a function that is called with constant values.

Write the function `call_myfn`, which calls `myfcn`.

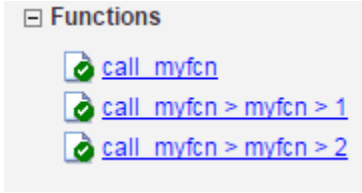
```
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, 'mode1');
y = myfcn(n, 'mode2');
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode,'mode1')
    y = n;
else
    y = -n;
```

```
end
end
```

Generate standalone C code. For example, generate a static library. Enable the code generation report.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you see two function specializations for `call_myfcn`.



The code generator creates `call_myfcn>myfcn>1` for `mode` with a value of `'mode1'`. It creates `call_myfcn>myfcn>2` for `mode` with a value of `'mode2'`.

In the generated C code, you see the specializations `my_fcn` and `b_my_fcn`.

```
static double b_myfcn(double n)
{
  return -n;
}

static double myfcn(double n)
{
  return n;
}
```

To prevent the function specializations, instruct the code generator to ignore that values of the `mode` argument are constant.

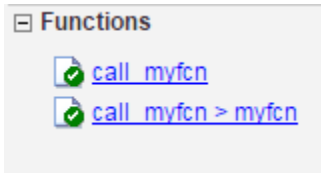
```
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, coder.ignoreConst('mode1'));
y = myfcn(n, coder.ignoreConst('mode2'));
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode,'mode1')
```

```
    y = n;
else
    y = -n;
end
end
```

Generate the C code.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you do not see multiple function specializations.



In the generated C code, you see one function for `my_fcn`.

Input Arguments

`expression` — Expression whose value is to be treated as a nonconstant
MATLAB expression

Definitions

Function Specialization

Version of a function in which an input type, size, complexity, or value is customized for a particular invocation of the function.

Function specialization produces efficient C code at the expense of code duplication. The code generation report shows all MATLAB function specializations that the code generator creates. However, the specializations might not appear in the generated C/C++ code due to later transformations or optimizations.

Tips

- For some recursive function calls, you can use `coder.ignoreConst` to force run-time recursion. See “Force Code Generator to Use Run-Time Recursion”.
- `coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations. It does not prevent other uses of the constant value during code generation.

See Also

`coder.inline`

Topics

“Force Code Generator to Use Run-Time Recursion”
“Compile-Time Recursion Limit Reached”
“Avoid Duplicate Functions in Generated Code”

Introduced in R2017a

coder.inline

Package: coder

Control inlining in generated code

Syntax

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

Description

`coder.inline('always')` forces inlining on page 2-108 of the current function in the generated code. Place the `coder.inline` directive inside the function to which it applies. The code generator does not inline entry-point functions, inline functions into `parfor` loops, or inline functions called from `parfor` loops.

`coder.inline('never')` prevents inlining of the current function in the generated code. Prevent inlining when you want to simplify the mapping between the MATLAB source code and the generated code. You can disable inlining for all functions at the command line by using the `-O disable:inline` option of the `codegen` command.

`coder.inline('default')` uses internal heuristics to determine whether to inline the current function. Usually, the heuristics produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

Examples

- “Prevent Function Inlining” on page 2-107
- “Use coder.inline in Control Flow Statements” on page 2-107

Prevent Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
  coder.inline('never');
  y = x;
end
```

Use coder.inline in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose that you want to generate code for a division function used by a system with limited memory. To optimize memory use in the generated code, the `inline_division` function 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('Cannot divide by 0');
end

y = dividend / divisor;
```

Definitions

Inlining

Technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code.

See Also

codegen

Topics

“Control Inlining”
“Optimization Strategies”

Introduced in R2011a

coder.isColumnMajor

Determine whether the current function or variable uses column-major layout

Syntax

```
coder.isColumnMajor
coder.isColumnMajor(arg)
```

Description

`coder.isColumnMajor` resolves as true in the generated code if the current function uses column-major array layout. Use the function as the expression in control flow (`if`, `else`, `switch`) statements.

`coder.isColumnMajor(arg)` resolves as true if the current variable uses column-major array layout.

Examples

Query Array Layout of a Function

To query the array layout of a function at compile time, use `coder.isColumnMajor` or `coder.isRowMajor`. This query can be useful for specializing your generated code when it involves row-major and column-major functions. For example, consider this function:

```
function S = addMatrixRouted(A,B)
 if coder.isRowMajor
     %execute this code if row major
     S = addMatrix_OptimizedForRowMajor(A,B);
 elseif coder.isColumnMajor
     %execute this code if column major
     S = addMatrix_OptimizedForColumnMajor(A,B);
 end
```

The function `addMatrixRouted` behaves differently depending on whether it uses row-major layout or column-major layout. The layout that the function uses, for example, can depend on whether it is called from a function that contains `coder.rowMajor` or `coder.columnMajor`. When `addMatrixRouted` uses row-major layout, it calls the `addMatrix_OptimizedForRowMajor` function, which has efficient memory access for row-major data. When the function uses column-major layout, it calls a version of the `addMatrix` function optimized for column-major data.

By using the query functions, the generated code for `addMatrixRouted` provides efficient memory access for either choice of array layout.

Query Array Layout of a Variable

Consider the function `bar`:

```
function bar
coder.columnMajor;
x = magic(3);
if coder.isColumnMajor(x)
    fprintf('This will always be displayed in generated code.\n');
else
    fprintf('This will never be displayed in generated code.\n');
end
end
```

Generate code:

```
codegen bar
```

To run the MEX function, enter:

```
bar_mex
```

Input Arguments

arg — Variable name

array variable

Variable to query for array layout.

Example: `coder.isColumnMajor(x);`

Limitations

- You cannot query the array layout of a structure field or property.

Tips

- The code generator uses column-major layout by default.
- Outside of code generation, `coder.isColumnMajor` is always true.
- If `coder.isColumnMajor` always resolves to true for your code, other branches in the `if` statement are ignored by the code generator. Otherwise, one instance of the current function is created for each array layout.

See Also

`coder.columnMajor` | `coder.isRowMajor` | `coder.rowMajor`

Topics

“Row-Major and Column-Major Array Layouts”
“Specify Array Layout in Functions and Classes”
“Code Design for Row-Major Array Layout”

Introduced in R2018a

coder.isRowMajor

Determine whether the current function or variable uses row-major layout

Syntax

```
coder.isRowMajor
coder.isRowMajor(arg)
```

Description

`coder.isRowMajor` resolves as true in the generated code if the current function uses row-major array layout. Use the function as the expression in control flow (`if`, `else`, `switch`) statements.

`coder.isRowMajor(arg)` resolves as true if the current variable uses row-major array layout.

Examples

Query Array Layout of a Function

To query the array layout of a function at compile time, use `coder.isRowMajor` or `coder.isColumnMajor`. This query can be useful for specializing your generated code when it involves row-major and column-major functions. For example, consider this function:

```
function S = addMatrixRouted(A,B)
 if coder.isRowMajor
     %execute this code if row major
     S = addMatrix_OptimizedForRowMajor(A,B);
 elseif coder.isColumnMajor
     %execute this code if column major
     S = addMatrix_OptimizedForColumnMajor(A,B);
 end
```

The function `addMatrixRouted` behaves differently depending on whether it uses row-major layout or column-major layout. The layout that the function uses, for example, can depend on whether it is called from a function that contains `coder.rowMajor` or `coder.columnMajor`. When `addMatrixRouted` uses row-major layout, it calls the `addMatrix_OptimizedForRowMajor` function, which has efficient memory access for row-major data. When the function uses column-major layout, it calls a version of the `addMatrix` function optimized for column-major data.

By using the query functions, the generated code for `addMatrixRouted` provides efficient memory access for either choice of array layout.

Query Array Layout of a Variable

Consider the function `foo`:

```
function foo
coder.rowMajor;
x = magic(3);
if coder.isRowMajor(x)
    fprintf('This will always be displayed in generated code.\n');
else
    fprintf('This will never be displayed in generated code.\n');
end
end
```

Generate code:

```
codegen foo
```

To run the MEX function, enter:

```
foo_mex
```

Input Arguments

`arg` — Variable name

array variable

Variable to query for array layout.

Example: `coder.isRowMajor(x);`

Limitations

- You cannot query the array layout of a structure field or property.

Tips

- Outside of code generation, `coder.isRowMajor` is always false.
- If `coder.isRowMajor` always resolves to true for your code, other branches in the `if` statement are ignored by the code generator. Otherwise, one instance of the current function is created for each array layout.

See Also

`coder.columnMajor` | `coder.isColumnMajor` | `coder.rowMajor`

Topics

“Row-Major and Column-Major Array Layouts”
“Specify Array Layout in Functions and Classes”
“Code Design for Row-Major Array Layout”

Introduced in R2018a

coder.load

Load compile-time constants from MAT-file or ASCII file into caller workspace

Syntax

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

Description

`S = coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`S = coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

`S = coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

`S = coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`S = coder.load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`S = coder.load(filename,'-mat',var1,...,varN)` treats `filename` as a MAT-file and loads only the specified variables from the file.

`S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)` treats `filename` as a MAT-file and loads only the variables that match the specified regular expressions.

Examples

Load compile-time constants from MAT-file

Generate code for a function `edgeDetect1` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect1` uses `coder.load` to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];

save sobel.mat k
```

Write the function `edgeDetect1`.

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat','k');
H = conv2(double(originalImage),S.k, 'same');
V = conv2(double(originalImage),S.k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

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

Generate a static library for `edgeDetect1`.

```
codegen -report -config cfg edgeDetect1
```

`codegen` generates C code in the `codegen\lib\edgeDetect1` folder.

Load compile-time constants from ASCII file

Generate code for a function `edgeDetect2` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect2` uses `coder.load` to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];
save sobel.dat k -ascii
```

Write the function `edgeDetect2`.

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = coder.load('sobel.dat');
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

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

Generate a static library for `edgeDetect2`.

```
codegen -report -config cfg edgeDetect2
```

`codegen` generates C code in the `codegen\lib\edgeDetect2` folder.

Input Arguments

`filename` — Name of file

character vector | string scalar

Name of file. `filename` must be a compile-time constant.

`filename` can include a file extension and a full or partial path. If `filename` has no extension, `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

Example: `'myFile.mat'`

`var1,...,varN` — Names of variables to load
character vector | string scalar

Names of variables, specified as one or more character vectors or string scalars. Each variable name must be a compile-time constant. Use the * wildcard to match patterns.

Example: `coder.load('myFile.mat','A*')` loads all variables in the file whose names start with `A`.

`expr1,...,exprN` — Regular expressions indicating which variables to load
character vector | string scalar

Regular expressions indicating which variables to load specified as one or more character vectors or string scalars. Each regular expression must be a compile-time constant.

Example: `coder.load('myFile.mat', '-regexp', '^A')` loads only variables whose names begin with `A`.

Output Arguments

`S` — Loaded variables or data
structure array | m-by-n array

If `filename` is a MAT-file, `S` is a structure array.

If `filename` is an ASCII file, `S` is an m-by-n array of type `double`. m is the number of lines in the file and n is the number of values on a line.

Limitations

- Arguments to `coder.load` must be compile-time constants.
- The output `S` must be the name of a structure or array without any subscripting. For example, `S(i) = coder.load('myFile.mat')` is not allowed.
- You cannot use `save` to save workspace data to a file inside a function intended for code generation. The code generator does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save workspace data to a file.

Tips

- `coder.load` loads data at compile time, not at run time. If you are generating MEX code or code for Simulink simulation, you can use the MATLAB function `load` to load run-time values.
- If the MAT-file contains unsupported constructs, use `coder.load(filename,var1,...,varN)` to load only the supported constructs.
- If you generate code in a MATLAB Coder project, the code generator practices incremental code generation for the `coder.load` function. When the MAT-file or ASCII file used by `coder.load` changes, the software rebuilds the code.

See Also

`matfile` | `regexp` | `save`

Topics

“Regular Expressions” (MATLAB)

Introduced in R2013a

coder.loadDeepLearningNetwork

Load deep learning network model

Syntax

```
net = coder.loadDeepLearningNetwork(filename)
net = coder.loadDeepLearningNetwork(functionname)
net = coder.loadDeepLearningNetwork( ___ ,network_name)
```

Description

`net = coder.loadDeepLearningNetwork(filename)` loads a pretrained deep learning `SeriesNetwork` or `DAGNetwork` object saved in the `filename` MAT-file. `filename` must be a valid MAT-file existing on the MATLAB path containing a single `SeriesNetwork` or `DAGNetwork` object.

`net = coder.loadDeepLearningNetwork(functionname)` calls a function that returns a pretrained deep learning `SeriesNetwork` or `DAGNetwork` object. `functionname` must be the name of a function existing on the MATLAB path that returns a `SeriesNetwork` or `DAGNetwork` object.

`net = coder.loadDeepLearningNetwork( ___ ,network_name)` is the same as `net = coder.loadDeepLearningNetwork(filename)` with the option to name the C++ class generated from the network. `network_name` is a descriptive name for the network object saved in the MAT-file or pointed to by the function. The network name must be a `char` type that is a valid identifier in C++.

Use this function when generating code from a network object inference. This function generates a C++ class from this network. The class name is derived from the MAT-file name or the function name.

Examples

Generate CUDA Code from a MAT-File Containing the `AlexNet` Network

Use of the `coder.loadDeepLearningNetwork` function to load an `AlexNet` series network and generate CUDA code for this network.

Get the MAT-file containing the pretrained `AlexNet` network.

```
url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/alexnet/alexnet.mat';
websave('alexnet.mat',url);
```

Create an entry-point function `alexneteg` that uses the `coder.loadDeepLearningNetwork` function to load the `alexnet.mat` into the persistent `mynet` `SeriesNetwork` object.

```
function out = alexneteg(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('alexnet.mat', 'myalexnet');
end

out = predict(mynet,in);
```

The persistent object avoids reconstructing and reloading the network object during subsequent calls to the function to invoke the `predict` method on the input

The input layer of the pretrained `AlexNet` network accepts images of size `227x227x3`. Use the following lines of code to read an input image from a graphics file and resize it to `227x227`.

```
in = imread('peppers.png');
in = imresize(in,[227,227]);
```

Create a `coder.gpuConfig` configuration object for MEX code generation and set the target language to C++. On the configuration object, set `DeepLearningConfig` with `targetlib` as `'cudnn'`. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function. Use the `-config` option to pass the code configuration object.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -args {ones(227,227,3,'single')} -config cfg alexneteg
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the CUDA code for the entry-point function `alexneteg.cu`, header and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Code Generation for a `SeriesNetwork` Inference Loaded from a MATLAB Function

Use of the `coder.loadDeepLearningNetwork` function to load an `AlexNet` series network and generate CUDA code for this network.

Create an entry-point function `alexnetfun` that uses the `coder.loadDeepLearningNetwork` function to call the Deep Learning Toolbox toolbox function `alexnet`. This function returns a pretrained `AlexNet` network.

```
function out = alexnetfun(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('alexnet', 'myalexnet');
end

out = predict(mynet,in);
```

The persistent object avoids reconstructing and reloading the network object during subsequent calls to the function to invoke the `predict` method on the input

The input layer of the pretrained `AlexNet` network accepts images of size `227x227x3`. To read an input image from a graphics file and resize it to `227x227`, use the following lines of code:

```
in = imread('peppers.png');
in = imresize(in,[227,227]);
```

Create a `coder.gpuConfig` configuration object for MEX code generation and set the target language to C++. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function and the `-config` option to pass the code configuration object.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
```

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -args {ones(227,227,3,'single')} -config cfg alexnetfun
```

The `codegen` command places all the generated files in the `codegen` folder. It contains the CUDA code for the entry-point function `alexneteg.cu`, header, and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Input Arguments

`filename` — MAT file name

name

Specifies the name of the MAT-file containing the pretrained `SeriesNetwork` or `DAGNetwork` object.

Data Types: `string`

`functionname` — MATLAB function name

name

Specifies the name of the function that returns a pretrained `SeriesNetwork` or `DAGNetwork` object.

Data Types: `string`

`network_name` — Descriptive name

name

Descriptive name for the `SeriesNetwork` object saved in the MAT-file. It must be a `char` type that is a valid identifier in C++.

Data Types: `char`

Output Arguments

`net` — Network object

`SeriesNetwork` object | `DAGNetwork` object

Network inference, returned as a `SeriesNetwork` object or a `DAGNetwork` object.

Limitations

- `coder.loadDeepLearningNetwork` does not support loading MAT-files with multiple networks. The MAT-file must contain only the network to be loaded.

See Also

`cnncodegen` | `codegen`

Topics

“Load Pretrained Networks for Code Generation”
“Code Generation for Deep Learning Networks with MKL-DNN”
“Code Generation for Deep Learning Networks with ARM Compute Library”
“Code Generation for Deep Learning Networks with cuDNN” (GPU Coder)
“Code Generation for Deep Learning Networks with TensorRT” (GPU Coder)

Introduced in R2017b

coder.newtype

Package: coder

Create a `coder.Type` object

Syntax

```
t = coder.newtype(numeric_class, sz, variable_dims)
t = coder.newtype(numeric_class, sz, variable_dims, Name, Value)
t = coder.newtype('constant', value)
t = coder.newtype('struct', struct_fields, sz, variable_dims)
t = coder.newtype('cell', cells, sz, variable_dims)
t = coder.newtype('embedded.fi', numerictype, sz, variable_dims,
Name, Value)
t = coder.newtype(enum_value, sz, variable_dims)
t = coder.newtype(class_name)
t = coder.newtype('string')
```

Description

Note `coder.newtype` is an advanced function that you can use to control the `coder.Type` object. Consider using `coder.typeof` instead. `coder.typeof` creates a type from a MATLAB example.

`t = coder.newtype(numeric_class, sz, variable_dims)` creates a `coder.Type` object representing values of class `numeric_class` with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `variable_dims` is not specified, the dimensions of the type are fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to dimensions of the type that are not `1` or `0`, which are fixed.

`t = coder.newtype(numeric_class, sz, variable_dims, Name, Value)` creates a `coder.Type` object with additional options specified by one or more Name, Value pair arguments.

`t = coder.newtype('constant', value)` creates a `coder.Constant` object representing a single value. Use this type to specify a value that must be treated as a constant in the generated code.

`t = coder.newtype('struct', struct_fields, sz, variable_dims)` creates a `coder.StructType` object for an array of structures that has the same fields as the scalar structure `struct_fields`. The structure array type has the size specified by `sz` and variable-size dimensions specified by `variable_dims`.

`t = coder.newtype('cell', cells, sz, variable_dims)` creates a `coder.CellType` object for a cell array that has the cells and cell types specified by `cells`. The cell array type has the size specified by `sz` and variable-size dimensions specified by `variable_dims`. You cannot change the number of cells or specify variable-size dimensions for a heterogeneous cell array.

`t = coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)` creates a `coder.FiType` object representing a set of fixed-point values with `numerictype` and additional options specified by one or more Name, Value pair arguments.

`t = coder.newtype(enum_value, sz, variable_dims)` creates a `coder.Type` object representing a set of enumeration values of class `enum_value`.

`t = coder.newtype(class_name)` creates a `coder.ClassType` object for an object of the class `class_name`.

`t = coder.newtype('string')` creates a type for a string scalar. A string scalar contains one piece of text represented as a character vector. To specify the size of the character vector and whether the second dimension is variable-size, create a type for the character vector and assign it to the `Value` property of the string scalar type. For example, `t.Properties.Value = coder.newtype('char',[1 10], [0 1])` specifies that the character vector inside the string scalar is variable-size with an upper bound of 10.

Input Arguments

`numeric_class`

Class of the set of values represented by the type object.

`struct_fields`

Scalar structure used to specify the fields in a new structure type.

`cells`

Cell array of `coder.Type` objects that specify the types of the cells in a new cell array type.

`sz`

Size vector specifying each dimension of type object. `sz` cannot change the number of cells for a heterogeneous cell array.

Default: `[1 1]`

`class_name`

Name of class from which to create the `coder.ClassType`, specified as a character vector or string scalar. `class_name` must be the name of a value class.

`variable_dims`

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false). You cannot specify variable-size dimensions for a heterogeneous cell array.

Default: `true` for dimensions for which `sz` specifies an upper bound of `inf`; `false` for all other dimensions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`complex`

Set `complex` to `true` to create a `coder.Type` object that can represent complex values. The type must support complex data.

Default: false

`fimath`

Specify local `fimath`. If `fimath` is not specified, uses default fimath values.

Use only with `t=`coder.newtype('embedded.fi', numerictype,`sz`, `variable_dims`, Name, Value).

`sparse`

Set `sparse` to `true` to create a `coder.Type` object representing sparse data. The type must support sparse data.

Not for use with `t=`coder.newtype('embedded.fi', numerictype,`sz`, `variable_dims`, Name, Value)

Default: false

Output Arguments

`t`

New `coder.Type` object.

Examples

Create a type for use in code generation.

```
t=coder.newtype('double',[2 3 4],[1 1 0])
% Returns double :2x:3x4
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size

```
coder.newtype('double',[inf,3])
%   returns double:inf x 3

coder.newtype('double', [inf, 3], [1 0])
%   also returns double :inf x3
%   ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size with an upper bound of 3

```
coder.newtype('double', [inf,3],[0 1])
%  returns double :inf x :3
%  ':' indicates variable-size dimensions
```

Create a structure type to use in code generation.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
coder.newtype('struct',struct('a',ta,'b',tb))
% returns struct 1x1
%            a: int8 1x1
%            b: double :1x:2
% ':' indicates variable-size dimensions
```

Create a cell array to use in code generation.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
coder.newtype('cell',{ta, tb})
% returns 1x2 heterogeneous cell
%            f0: 1x1 int8
%            f1: :1x:2 double
% ':' indicates variable-size dimensions
```

Create a new constant type to use in code generation.

```
k = coder.newtype('constant', 42);
% Returns
% k =
%
% coder.Constant
%       42
```

Create a `coder.EnumType` object using the name of an existing MATLAB enumeration.

1. Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

2. Create a `coder.EnumType` object from this enumeration.

```
t = coder.newtype('MyColors');
```

Create a fixed-point type for use in code generation. The fixed-point type uses default fimath values.

```
t = coder.newtype('embedded.fi',...
   numerictype(1, 16, 15), [1 2])

t =
% Returns
% coder.FiType
%   1x2 embedded.fi
%     DataTypeMode: Fixed-point: binary point scaling
%     Signedness: Signed
%     WordLength: 16
%     FractionLength: 15
```

Create a type for an object to use in code generation.

1. Create this value class:

```
classdef mySquare
    properties
        side;
    end
    methods
        function obj = mySquare(val)
        end
    end
    if nargin > 0
        obj.side = val;
    end
    function a = calcarea(obj)
    a = obj.side * obj.side;
```

```
    end
end
end
```

2. Create a type for an object that has the same properties as `mySquare`.

```
t = coder.newtype('mySquare')
```

3. Change the type of the property `side`.

```
t.Properties.side = coder.typeof(int8(3))
```

```
t =

coder.ClassType
   1×1 mySquare
      side: 1×1 int8
```

Create a type for a string scalar for use in code generation.

1. Create the string scalar type.

```
t = coder.newtype('string');
```

2. Specify the size.

```
t.Properties.Value = coder.newtype('char',[1, 10])
```

```
t =

coder.ClassType
   1×1 string -> redirected to -> coder.internal.string
      Value: 1×10 char
```

3. Make the string variable-size with an upper bound of 10.

```
t.Properties.Value = coder.newtype('char',[1, 10], [0, 1])
```

4. Make the string variable-size with no upper bound.

```
t.Properties.Value = coder.newtype('char',[1, inf])
```

Limitations

- For sparse matrices, `coder.newtype` drops upper bounds for variable-size dimensions.

Tips

- `coder.newtype` fixes the size of a singleton dimension unless the `variable_dims` argument explicitly specifies that the singleton dimension has a variable size.

  For example, the following code specifies a 1-by-:10 double. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

  ```
  t = coder.newtype('double',[1 10],1)
  ```

  By contrast, the following code specifies a :1-by-:10 double. Both dimensions have a variable size.

  ```
  t = coder.newtype('double',[1 10],[1 1])
  ```

  Note For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

Alternatives

`coder.typeof`

See Also

`codegen` | `coder.ArrayType` | `coder.CellType` | `coder.ClassType` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.resize`

Introduced in R2011a

coder.nullcopy

Package: coder

Declare uninitialized variables in code generation

Syntax

```
X = coder.nullcopy(A)
```

Description

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. The function preallocates memory for `X` without incurring the overhead of initializing memory. In code generation, the `coder.nullcopy` function declares uninitialized variables. In MATLAB, `coder.nullcopy` returns the input such that X is equal to A.

If X is a structure containing variable-sized arrays, then you must assign the size of each array. `coder.nullcopy` does not copy sizes of arrays or nested arrays from its argument to its result.

Note Before you use `X` in a function or a program, ensure that the data in `X` is completely initialized . Declaring a variable through `coder.nullcopy` without assigning all the elements of the variable results in nondeterministic program behavior. For more information, see “How to Eliminate Redundant Copies by Defining Uninitialized Variables”.

Examples

Declare Variables for Optimized Initialization

Declare variable `X` as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo %#codegen

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

Using `coder.nullcopy` with `zeros` lets you specify the size of vector `X` without initializing each element to zero.

Input Arguments

`A` — Variable to copy

scalar | vector | matrix | multidimensional array

Variable to copy, specified as a scalar, vector, matrix, or multidimensional array.

Example: coder.nullcopy(A);

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`
Complex Number Support: Yes

Limitations

You cannot use `coder.nullcopy` on sparse matrices, structures, cell arrays, or classes that contain sparse matrices.

See Also

Topics

“Eliminate Redundant Copies of Variables in Generated Code”

Introduced in R2011a

coder.opaque

Declare variable in generated code

Syntax

```
y = coder.opaque(type)
y = coder.opaque(type,value)
y = coder.opaque( ___ ,'Size',Size)
y = coder.opaque( ___ ,'HeaderFile',HeaderFile)
```

Description

`y = coder.opaque(type)` declares a variable `y` with the specified type and no initial value in the generated code.

- `y` can be a variable or a structure field.
- MATLAB code cannot set or access `y`, but external C functions can accept `y` as an argument.
- `y` can be an:
  - Argument to `coder.rref`, `coder.wref`, or `coder.ref`
  - Input or output argument to `coder.ceval`
  - Input or output argument to a user-written MATLAB function
  - Input to a subset of MATLAB toolbox functions supported for code generation
- Assignment from `y` declares another variable with the same type in the generated code. For example:

  ```
  y = coder.opaque('int');
  z = y;
  ```

  declares a variable `z` of type `int` in the generated code.
- You can assign `y` from another variable declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types.

- You can compare `y` to another variable declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types.

`y = coder.opaque(type,value)` specifies the type and initial value of `y`.

`y = coder.opaque( ___ ,'Size',Size)` specifies the size, in bytes, of `y`. You can specify the size with any of the previous syntaxes.

`y = coder.opaque( ___ ,'HeaderFile',HeaderFile)` specifies the header file that contains the type definition. The code generator produces the `#include` statement for the header file where the statement is required in the generated code. You can specify the header file with any of the previous syntaxes.

Examples

Declare Variable Specifying Initial Value

Generate code for a function `valtest` which returns `1` if the call to `myfun` is successful. This function uses `coder.opaque` to declare a variable `x1` with type `int` and initial value `0`. The assignment `x2 = x1` declares `x2` to be a variable with the type and initial value of `x1`.

Write a function `valtest`.

```
function y = valtest
%codegen
%declare x1 to be an integer with initial value '0'
x1 = coder.opaque('int','0');
%Declare x2 to have same type and initial value as x1
x2 = x1;
x2 = coder.ceval('myfun');
%test the result of call to 'myfun' by comparing to value of x1
if x2 == x1
  y = 0;
else
  y = 1;
```

```
end
end
```

Declare Variable Specifying Initial Value and Header File

Generate code for a MATLAB function `filetest` which returns its own source code using `fopen/fread/fclose`. This function uses `coder.opaque` to declare the variable that stores the file pointer used by `fopen/fread/fclose`. The call to `coder.opaque` declares the variable `f` with type `FILE *`, initial value `NULL`, and header file `<stdio.h>`.

Write a MATLAB function `filetest`.

```
function buffer = filetest
%#codegen

% Declare 'f' as an opaque type 'FILE *' with initial value 'NULL"
%Specify the header file that contains the type definition of 'FILE *';

f = coder.opaque('FILE *', 'NULL','HeaderFile','<stdio.h>');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));

% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
    % By default, MATLAB converts constant values
    % to doubles in generated code
    % so explicit type conversion to int32 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 character vector
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;
```

Compare Variables Declared Using `coder.opaque`

Compare variables declared using `coder.opaque` to test for successfully opening a file.

Use `coder.opaque` to declare a variable `null` with type `FILE *` and initial value `NULL`.

```
null = coder.opaque('FILE *', 'NULL', 'HeaderFile', '<stdio.h>');
```

Use assignment to declare another variable `ftmp` with the same type and value as `null`.

```
ftmp = null;
ftmp = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

Compare the variables.

```
if ftmp == null
        %error condition
end
```

Cast to and from Types of Variables Declared Using `coder.opaque`

This example shows how to cast to and from types of variables that are declared using `coder.opaque`. The function `castopaque` calls the C run-time function `strncmp` to compare at most `n` characters of the strings `s1` and `s2`. `n` is the number of characters in the shorter of the strings. To generate the correct C type for the `strncmp` input `nsizet`, the function casts `n` to the C type `size_t` and assigns the result to `nsizet`. The function uses `coder.opaque` to declare `nsizet`. Before using the output `retval` from `strncmp`, the function casts `retval` to the MATLAB type `int32` and stores the results in `y`.

Write this MATLAB function:

```
function y = castopaque(s1,s2)

%  <0 - the first character that does not match has a lower value in s1 than in s2
%   0 - the contents of both strings are equal
%  >0 - the first character that does not match has a greater value in s1 than in s2
%
%#codegen
```

```
coder.cinclude('<string.h>');
n = min(numel(s1), numel(s2));

% Convert the number of characters to compare to a size_t

nsizet = cast(n,'like',coder.opaque('size_t','0'));

% The return value is an int
retval = coder.opaque('int');
retval = coder.ceval('strncmp', cstr(s1), cstr(s2), nsizet);

% Convert the opaque return value to a MATLAB value
y = cast(retval, 'int32');

%--------------
function sc = cstr(s)
% NULL terminate a MATLAB character vector for C
sc = [s, char(0)];
```

Generate the MEX function.

```
codegen castopaque -args {blanks(3), blanks(3)} -report
```

Call the MEX function with inputs `'abc'` and `'abc'`.

```
castopaque_mex('abc','abc')
```

```
ans =

           0
```

The output is `0` because the strings are equal.

Call the MEX function with inputs `'abc'` and `'abd'`.

```
castopaque_mex('abc','abd')
```

```
ans =

          -1
```

The output is `-1` because the third character `d` in the second string is greater than the third character `c` in the first string.

Call the MEX function with inputs `'abd'` and `'abc'`.

```
castopaque_mex('abd','abc')

ans =

           1
```

The output is `1` because the third character `d` in the first string is greater than the third character `c` in the second string.

In the MATLAB workspace, you can see that the type of `y` is `int32`.

Declare Variable Specifying Initial Value and Size

Declare `y` to be a 4-byte integer with initial value 0.

```
y = coder.opaque('int','0', 'Size', 4);
```

Input Arguments

`type` — Type of variable
character vector | string scalar

Type of variable in generated code. `type` must be a compile-time constant. The type must be a:

- Built-in C data type or a type defined in a header file
- C type that supports copy by assignment
- Legal prefix in a C declaration

Example: `'FILE *'`

`value` — Initial value of variable
character vector | string scalar

Initial value of variable in generated code. `value` must be a compile-time constant. Specify a C expression not dependent on MATLAB variables or functions.

If you do not provide the initial value in `value`, initialize the value of the variable before using it. To initialize a variable declared using `coder.opaque`:

- Assign a value from another variable with the same type declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`.
- Assign a value from an external C function.
- Pass the address of the variable to an external function using `coder.wref`.

Specify a `value` that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results.

Example: `'NULL'`

`Size` — Size of variable
integer

Number of bytes for the variable in the generated code, specified as an integer. If you do not specify the size, the size of the variable is 8 bytes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HeaderFile` — Name of header file
character vector | string scalar

Name of header file that contains the definition of `type`. `HeaderFile` must be a compile-time constant.

For a system header file, use angle brackets.

Example: `'<stdio.h>'` generates `#include <stdio.h>`

For an application header file, use double quotes.

Example: `'"foo.h"'` generates `#include "foo.h"`

If you omit the angle brackets or double quotes, the code generator produces double quotes.

Example: `'foo.h'` generates `#include "foo.h"`

Specify the include path in the build configuration parameters.

Example: `cfg.CustomInclude = 'c:\myincludes'`

Tips

- Specify a `value` that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results. For example, the following `coder.opaque` declaration can produce unexpected results.

  ```
  y = coder.opaque('int', '0.2')
  ```

- `coder.opaque` declares the type of a variable. It does not instantiate the variable. You can instantiate a variable by using it later in the MATLAB code. In the following example, assignment of `fp1` from `coder.ceval` instantiates `fp1`.

  ```
  % Declare fp1 of type FILE *
  fp1 = coder.opaque('FILE *');
  %Create the variable fp1
  fp1 = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
  ```

- In the MATLAB environment, `coder.opaque` returns the value specified in `value`. If `value` is not provided, it returns an empty character vector.
- You can compare variables declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types. The following example demonstrates how to compare these variables. “Compare Variables Declared Using coder.opaque” on page 2-139
- To avoid multiple inclusions of the same header file in generated code, enclose the header file in the conditional preprocessor statements `#ifndef` and `#endif`. For example:

  ```
  #ifndef MyHeader_h
  #define MyHeader_h
  <body of header file>
  #endif
  ```

- You can use the MATLAB `cast` function to cast a variable to or from a variable that is declared using `coder.opaque`. Use `cast` with `coder.opaque` only for numeric types.

  To cast a variable declared by `coder.opaque` to a MATLAB type, you can use the `B = cast(A,type)` syntax. For example:

  ```
  x = coder.opaque('size_t','0');
  x1 = cast(x, 'int32');
  ```

  You can also use the `B = cast(A,'like',p)` syntax. For example:

```
x = coder.opaque('size_t','0');
x1 = cast(x, 'like', int32(0));
```

To cast a MATLAB variable to the type of a variable declared by `coder.opaque`, you must use the `B = cast(A,'like',p)` syntax. For example:

```
x = int32(12);
x1 = coder.opaque('size_t', '0');
x2 = cast(x, 'like', x1));
```

Use `cast` with `coder.opaque` to generate the correct data types for:

- Inputs to C/C++ functions that you call using `coder.ceval`.
- Variables that you assign to outputs from C/C++ functions that you call using `coder.ceval`.

Without this casting, it is possible to receive compiler warnings during code generation.

See Also

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

Topics

“Specify Build Configuration Parameters”
“Call C/C++ Code from MATLAB Code”

Introduced in R2011a

coder.ref

Indicate data to pass by reference

Syntax

```
coder.ref(arg)
```

Description

`coder.ref(arg)` indicates that `arg` is an expression or variable to pass by reference to an external C/C++ function. Use `coder.ref` inside a `coder.ceval` call only. The C/C++ function can read from or write to the variable passed by reference. Use a separate `coder.ref` construct for each argument that you pass by reference to the function.

See also `coder.rref` and `coder.wref`.

Examples

Pass Scalar Variable by Reference

Consider the C function `addone` that returns the value of an input plus one:

```
double addone(double* p) {
  return *p + 1;
}
```

The C function defines the input variable `p` as a pointer to a double.

Pass the input by reference to `addone`:

```
...
y = 0;
u = 42;
```

```
y = coder.ceval('addone', coder.ref(u));
...
```

Pass Multiple Arguments by Reference

```
...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.ref(u), coder.ref(v));
...
```

Pass Class Property by Reference

```
...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.ref(x.prop));
...
```

Pass a Structure by Reference

To indicate that the structure type is defined in a C header file, use `coder.cstructname`.

Suppose that you have the C function `incr_struct`. This function reads from and writes to the input argument.

```
#include "MyStruct.h"

void incr_struct(struct MyStruct *my_struct)
{
  my_struct->f1 =  my_struct->f1 + 1;
  my_struct->f2 =  my_struct->f2 + 1;
}
```

The C header file, `MyStruct.h`, defines a structure type named `MyStruct`:

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
```

```
    double f1;
    double f2;
} MyStruct;

void incr_struct(struct MyStruct *my_struct);

#endif
```

In your MATLAB function, pass a structure by reference to `incr_struct`. To indicate that the structure type for `s` has the name `MyStruct` that is defined in the C header file `MyStruct.h`, use `coder.cstructname`.

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','incr_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
coder.ceval('incr_struct', coder.ref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

Pass Structure Field by Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.ref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.ref(a.b(3,4).c(2).u));
...
```

Input Arguments

`arg` — Argument to pass by reference
scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`
Complex Number Support: Yes

Limitations

- You cannot pass these data types by reference:
  - Class or System object
  - Cell array or index into a cell array
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- If `arg` is an array, then `coder.ref(arg)` provides the address of the first element of the array. The `coder.ref(arg)` function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  ```
  coder.ceval('myFun',coder.ref(arg),int32(numel(arg));
  ```
- When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.
- In MATLAB, `coder.ref` results in an error. To parameterize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
- You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.rref` | `coder.wref` | `numel`

Topics

“Call C/C++ Code from MATLAB Code”

Introduced in R2011a

coder.resize

Package: coder

Resize a `coder.Type` object

Syntax

```
t_out = coder.resize(t, sz, variable_dims)
t_out = coder.resize(t, sz)
t_out = coder.resize(t,[],variable_dims)
t_out = coder.resize(t, sz, variable_dims, Name, Value)
t_out = coder.resize(t, 'sizelimits', limits)
```

Description

`t_out = coder.resize(t, sz, variable_dims)` returns a modified copy of `coder.Type` `t` with upper-bound size `sz`, and variable dimensions `variable_dims`. If `variable_dims` or `sz` are scalars, the function applies them to all dimensions of `t`. By default, `variable_dims` does not apply to dimensions where `sz` is `0` or `1`, which are fixed. Use the 'uniform' option to override this special case. `coder.resize` ignores `variable_dims` for dimensions with size `inf`. These dimensions are always variable size. `t` can be a cell array of types, in which case, `coder.resize` resizes all elements of the cell array.

`t_out = coder.resize(t, sz)` resizes `t` to have size `sz`.

`t_out = coder.resize(t,[],variable_dims)` changes `t` to have variable dimensions `variable_dims` while leaving the size unchanged.

`t_out = coder.resize(t, sz, variable_dims, Name, Value)` resizes `t` using additional options specified by one or more Name, Value pair arguments.

`t_out = coder.resize(t, 'sizelimits', limits)` resizes `t` with dimensions becoming variable based on the `limits` vector. When the size `S` of a dimension is greater than or equal to the first threshold defined in `limits`, the dimension becomes variable

size with upper bound `S`. When the size `S` of a dimension is greater than or equal to the second threshold defined in `limits`, the dimension becomes unbounded variable size.

Input Arguments

`limits`

Two-element vector (or a scalar-expanded, one-element vector) of variable-sizing thresholds. If the size `sz` of a dimension of `t` is greater than or equal to the first threshold, the dimension becomes variable size with upper bound `sz`. If the size `sz` of a dimension of `t` is greater than or equal to the second threshold, the dimension becomes unbounded variable size.

`sz`

New size for `coder.Type` object, `t_out`

`t`

`coder.Type` object that you want to resize. If `t` is a `coder.CellType` object, the `coder.CellType` object must be homogeneous.

`variable_dims`

Specify whether each dimension of *t_out* is fixed or variable size.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`recursive`

Setting `recursive` to `true` resizes *t* and all types contained within it.

Default: false

`uniform`

Setting `uniform` to `true` resizes *t* but does not apply the heuristic for dimensions of size one.

Default: false

Output Arguments

`t_out`

Resized `coder.Type` object

Examples

Change a fixed-size array to a bounded, variable-size array.

```
t = coder.typeof(ones(3,3))
% t is     3x3
coder.resize(t, [4 5], 1)
% returns :4 x :5
% ':' indicates variable-size dimensions
```

Change a fixed-size array to an unbounded, variable-size array.

```
t = coder.typeof(ones(3,3))
% t is 3x3
coder.resize(t, inf)
% returns :inf x :inf
% ':' indicates variable-size dimensions
% 'inf' indicates unbounded dimensions
```

Resize a structure field.

```
ts = coder.typeof(struct('a', ones(3, 3)))
% returns field a as 3x3
coder.resize(ts, [5, 5], 'recursive', 1)
% returns field as 5x5
```

Resize a cell array.

```
tc = coder.typeof({1 2 3})
% returns 1x3 cell array
coder.resize(tc, [5, 5], 'recursive', 1)
% returns cell array as 5x5
```

Make a fixed-sized array variable size based on bounded and unbounded thresholds.

```
t = coder.typeof(ones(100,200))
% t is 100x200
coder.resize(t,'sizelimits', [99 199])
% returns :100x:inf
% ':' indicates variable-size dimensions
% :inf is unbounded variable size
```

Limitations

- For sparse matrices, `coder.resize` drops upper bounds for variable-size dimensions.

See Also

`codegen` | `coder.newtype` | `coder.typeof`

Introduced in R2011a

coder.rowMajor

Specify row-major array layout for a function or class

Syntax

```
coder.rowMajor
```

Description

`coder.rowMajor` specifies row-major array layout for the data used by the current function in generated code. When placed in a class constructor, `coder.rowMajor` specifies row-major layout for data used by the class.

Examples

Specify Row-Major Array Layout for a Function

Specify row-major array layout for a function by inserting `coder.rowMajor` into the function body.

Suppose that `myFunction` is the top-level function of your code. Your application requires you to perform matrix addition with row-major array layout and matrix multiplication with column-major layout.

```
function S = myFunction(A,B)
%#codegen
% check to make sure inputs are valid
if size(A,1) ~= size(B,1) || size(A,2) ~= size(B,2)
    disp('Matrices must be same size.');
    return;
end
% make both matrices symmetric
B = B*B';
A = A*A';
```

```
% add matrices
S = addMatrix(A,B);
end
```

Write a function for matrix addition called `addMatrix`. Specify row-major layout for `addMatrix` by using `coder.rowMajor`.

```
function S = addMatrix(A,B)
%#codegen
S = zeros(size(A));
coder.rowMajor; % specify row-major array layout
S = A + B;
end
```

Generate code for `myFunction`. Use the `codegen` command.

```
codegen myFunction -args {ones(10,20),ones(10,20)} -config:lib -launchreport
```

The code generator produces code for `addMatrix` that uses row-major array layout. However, the matrix multiplication from the top-level function uses the default layout, column-major.

Tips

- To specify row-major array layout for all the functions in your generated code, use the `codegen -rowmajor` option.
- Other functions called from within a row-major function inherit the row-major specification. However, if one of the called functions has its own distinct `coder.columnMajor` call, the code generator changes the array layout accordingly. If a row-major function and a column-major function call the same function, which does not have its own array layout specification, the code generator produces a row-major version and column-major version of the function.
- `coder.rowMajor` is ignored outside of code generation.

See Also

`coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor`

Topics

“Row-Major and Column-Major Array Layouts”

“Generate Code That Uses Row-Major Array Layout”
“Specify Array Layout in Functions and Classes”
“Generate Code That Uses N-Dimensional Indexing”

Introduced in R2018a

coder.rref

Indicate read-only data to pass by reference

Syntax

```
coder.rref(arg)
```

Description

`coder.rref(arg)` indicates that `arg` is a read-only expression or variable to pass by reference to an external C/C++ function. Use `coder.rref` only inside a `coder.ceval` call.

The `coder.rref` function can enable the code generator to optimize the generated code. Because the external function is assumed to not write to `coder.rref(arg)`, the code generator can perform optimizations such as expression folding on assignments to `arg` that occur before and after the `coder.ceval` call. Expression folding is the combining of multiple operations into one statement to avoid the use of temporary variables and improve code performance.

Note The code generator assumes that the memory that you pass with `coder.rref(arg)` is read-only. To avoid unpredictable results, the C/C++ function must not write to this variable.

See also `coder.ref` and `coder.wref`.

Examples

Pass Scalar Variable as a Read-Only Reference

Consider the C function `addone` that returns the value of a constant input plus one:

```
double addone(const double* p) {
  return *p + 1;
}
```

The C function defines the input variable `p` as a pointer to a constant double.

Pass the input by reference to `addone`:

```
...
y = 0;
u = 42;
y = coder.ceval('addone', coder.rref(u));
...
```

Pass Multiple Arguments as a Read-Only Reference

```
...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.rref(u), coder.rref(v));
...
```

Pass Class Property as a Read-Only Reference

```
...
x = myClass;
x.prop = 1;
y = coder.ceval('foo', coder.rref(x.prop));
...
```

Pass Structure as a Read-Only Reference

To indicate that the structure type is defined in a C header file, use `coder.cstructname`.

Suppose that you have the C function `use_struct`. This function reads from the input argument but does not write to it.

```
#include "MyStruct.h"

double use_struct(const struct MyStruct *my_struct)
{
```

```
  return my_struct->f1 + my_struct->f2;
}
```

The C header file, `MyStruct.h`, defines a structure type named `MyStruct`:

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

double use_struct(const struct MyStruct *my_struct);

#endif
```

In your MATLAB function, pass a structure as a read-only reference to `use_struct`. To indicate that the structure type for `s` has the name `MyStruct` that is defined in the C header file `MyStruct.h`, use `coder.cstructname`.

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','use_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
y = coder.ceval('use_struct', coder.rref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

Pass Structure Field as a Read-Only Reference

```
...
s = struct('s1', struct('a', [0 1]));
y = coder.ceval('foo', coder.rref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.rref(a.b(3,4).c(2).u));
...
```

Input Arguments

`arg` — Argument to pass by reference

scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`
Complex Number Support: Yes

Limitations

- You cannot pass these data types by reference:
  - Class or System object
  - Cell array or index into a cell array
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- If `arg` is an array, then `coder.rref(arg)` provides the address of the first element of the array. The `coder.rref(arg)` function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  ```
  coder.ceval('myFun',coder.rref(arg),int32(numel(arg));
  ```

- When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.
- In MATLAB, `coder.rref` results in an error. To parametrize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
- You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.wref`

Topics

“Call C/C++ Code from MATLAB Code”

Introduced in R2011a

coder.runTest

Run test replacing calls to MATLAB functions with calls to MEX functions

Syntax

```
coder.runTest(test,fcn)
coder.runTest(test,fcns,mexfcn)
coder.runTest(test,mexfile)
```

Description

`coder.runTest(test,fcn)` runs `test` replacing calls to `fcn` with calls to the compiled version of `fcn`. `test` is the file name for a MATLAB function, script, or class-based unit test that calls the MATLAB function `fcn`. The compiled version of `fcn` must be in a MEX function that has the default name. The default name is the name specified by `fcn` followed by `_mex`.

`coder.runTest(test,fcns,mexfcn)` replaces calls to the specified MATLAB functions with calls to the compiled versions of the functions. The MEX function `mexfcn` must contain the compiled versions of all of the specified MATLAB functions.

`coder.runTest(test,mexfile)` replaces a call to a MATLAB function with a call to the compiled version of the function when the compiled version of the function is in `mexfile`. `mexfile` includes the platform-specific file extension. If `mexfile` does not contain the compiled version of a function, `coder.runTest` runs the original MATLAB function. If you do not want to specify the individual MATLAB functions to replace, use this syntax.

Examples

Run Test File Replacing One Function

Use `coder.runTest` to run a test file. Specify replacement of one MATLAB function with the compiled version. You do not provide the name of the MEX function that contains the

compiled version. Therefore, `coder.runTest` looks for a MEX function that has the default name.

In a local, writable folder, create a MATLAB function, `myfun`.

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

In the same folder, create a test function, `mytest1`, that calls `myfun`.

```
function mytest1
c = myfun(10,20);
disp(c);
end
```

Run the test function in MATLAB.

```
mytest1
```

```
    30
```

Generate a MEX function for `myfun`.

```
codegen myfun -args {0,0}
```

In the current folder, `codegen` generates a MEX function that has the default name, `myfun_mex`.

Run `coder.runTest`. Specify that you want to run the test file `mytest1`. Specify replacement of `myfun` with the compiled version in `myfun_mex`.

```
coder.runTest('mytest1','myfun')
```

```
    30
```

The results are the same as when you run `mytest1` at the MATLAB command line.

Replace Multiple Functions That You Specify

Use `coder.runTest` to run a test file. Specify replacement of two functions with calls to the compiled versions. Specify the MEX function that contains the compiled versions of the functions.

In a local writable folder, create a MATLAB function, `myfun1`.

```
function y = myfun1(u) %#codegen
y = u;
end
```

In the same folder, create another MATLAB function, `myfun2`.

```
function y = myfun2(u, v) %#codegen
y = u + v;
end
```

In the same folder, create a test function that calls `myfun1` and `myfun2`.

```
function mytest2
c1 = myfun1(10);
disp(c1)
c2 = myfun2(10,20);
disp(c2)
end
```

Run the test function.

```
mytest2
```

```
    10

    30
```

Generate a MEX function for `myfun1` and `myfun2`. Use the `-o` option to specify the name of the generated MEX function.

```
codegen -o mymex  myfun1 -args {0} myfun2 -args {0,0}
```

Run `coder.runTest`. Specify that you want to run `mytest2`. Specify that you want to replace the calls to `myfun1` and `myfun2` with calls to the compiled versions in the MEX function `mymex`.

```
coder.runTest('mytest2',{'myfun1','myfun2'},'mymex')
```

```
    10

    30
```

The results are the same as when you run `mytest2` at the MATLAB command line.

Replace Functions That Have Compiled Versions in Specified MEX File

Use `coder.runTest` to run a test that replaces calls to MATLAB functions in the test with calls to the compiled versions. Specify the file name for the MEX function that contains the compiled versions of the functions.

In a local writable folder, create a MATLAB function, `myfun1`.

```
function y = myfun1(u) %#codegen
y = u;
end
```

In the same folder, create another MATLAB function, `myfun2`.

```
function y = myfun2(u, v) %#codegen
y = u + v;
end
```

In the same folder, create a test function that calls `myfun1` and `myfun2`.

```
function  mytest2
c1 = myfun1(10);
disp(c1)
c2 = myfun2(10,20);
disp(c2)
end
```

Run the test.

```
mytest2
```

```
    10

    30
```

Generate a MEX function for `myfun1` and `myfun2`. Use the `-o` option to specify the name of the generated MEX function.

```
codegen -o mymex  myfun1 -args {0} myfun2 -args {0,0}
```

Run `coder.runTest`. Specify that you want to run `mytest2`. Specify that you want to replace calls to functions called by `mytest2` with calls to the compiled versions in `mymex`.

Specify the complete MEX file name including the platform-specific extension. Use `mexext` to get the platform-specific extension.

```
coder.runTest('mytest2',['mymex.', mexext])
```

```
    10

    30
```

The results are the same as when you run `mytest2` at the MATLAB command line.

Run Class-Based Unit Test

Run `coder.runTest` with a class-based unit test.

Write the function `addOne`, which adds 1 to the input.

```
function y = addOne(x)
%#codegen
y = x + 1;
end
```

Write a classed-based unit test that verifies the value returned by `addOne` when the input is 1 and when the input is `pi`.

```
classdef TestAddOne < matlab.unittest.TestCase

    methods (Test)

        function reallyAddsOne(testCase)
            x = 1;
            y = addOne(x);
            testCase.verifyEqual(y,2);
        end

        function addsFraction(testCase)
            x = pi;
            y = addOne(x);
            testCase.verifyEqual(y,x+1);
        end
    end
end
```

Run the class-based unit test, replacing calls to `addOne` with calls to `addOne_mex`.

```
coder.runTest('TestAddOne', 'addOne')
```

```
Running TestAddOne
..
Done TestAddOne
__________


testbenchResult =

  1×2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.28516 seconds testing time.
```

Input Arguments

`test` — File name for test function, script, or class-based unit test

character vector | string scalar

File name for MATLAB function, script, or class-based unit test that calls the MATLAB functions for which you want to test the generated MEX. `coder.runTest` replaces the calls to the functions with calls to the generated MEX.

Example: `'mytest'`

`fcn` — Name of MATLAB function to replace

character vector | string scalar

Name of MATLAB function to replace when running test. `coder.runTest` replaces calls to this function with calls to the compiled version of this function.

Example: `'myfun'`

`fcns` — Names of MATLAB functions to replace
character vector | string scalar | cell array of character vectors

Names of MATLAB functions to replace when running test. `coder.runTest` replaces calls to these functions with calls to the compiled versions of these functions.

Specify one function as a character vector or a string scalar.

Example: `'myfun'`

Example: `"myfun"`

Specify multiple functions as a cell array of character vectors. Before using `coder.runTest`, compile these functions into a single MEX function.

Example: `{'myfun1', 'myfun2', 'myfun3'}`

`mexfcn` — MEX function name
character vector | string scalar

Name of a MEX function generated for one or more functions.

Generate this MEX function using the MATLAB Coder app or the `codegen` function.

Example: `'mymex'`

`mexfile` — MEX file name with extension
character vector

The file name and platform-specific extension of a MEX file for one or more functions. Use `mexext` to get the platform-specific MEX file extension.

Generate this MEX file using the MATLAB Coder app or the `codegen` function.

Example: `['myfunmex.', mexext]`

Data Types: `char`

Tips

- `coder.runTest` does not return outputs. To see test results, in the test, include code that displays the results.
- To compare MEX and MATLAB function behavior:

  - Run the test in MATLAB.
  - Use `codegen` to generate a MEX function.
  - Use `coder.runTest` to run the test replacing the call to the original function with a call to the compiled version in the MEX function.
- Before using `coder.runTest` to test multiple functions, compile the MATLAB functions into a single MEX function.
- If you use the syntax `coder.runTest(test, mexfile)`, use `mexext` to get the platform-specific MEX file name extension. For example:

  ```
  coder.runTest('my_test', ['mymexfun.', mexext])
  ```
- If errors occur during the test, you can debug the code using call stack information.
- You can combine MEX generation and testing in one step by running `codegen` with the `-test` option. For example, the following code generates a MEX function for `myfunction` and calls the test file `myfunction_test`, replacing calls to `myfunction` with calls to `myfunction_mex`.

  ```
  codegen myfunction -test myfunction_test
  ```

See Also

`codegen` | `coder` | `coder.getArgTypes`

Topics

“MATLAB Code Analysis”
“Author Class-Based Unit Tests in MATLAB” (MATLAB)
“Unit Test Generated Code with MATLAB Coder”

Introduced in R2012a

coder.screener

Determine if function is suitable for code generation

Syntax

```
coder.screener(fcn)
coder.screener(fcn_1,...,fcn_n )
```

Description

`coder.screener(fcn)` analyzes the entry-point MATLAB function, `fcn`. It identifies unsupported functions and language features as code generation compliance issues. It displays the code generation compliance issues in a report. If `fcn` calls other functions directly or indirectly that are not MathWorks® functions, `coder.screener` analyzes these functions. It does not analyze MathWorks functions. It is possible that `coder.screener` does not detect all code generation issues. Under certain circumstances, it is possible that `coder.screener` reports false errors.

`coder.screener(fcn_1,...,fcn_n )` analyzes entry-point functions (`fcn_1,...,fcn_n`).

Input Arguments

`fcn`

Name of entry-point MATLAB function that you want to analyze. Specify as a character vector or a string scalar.

`fcn_1,...,fcn_n`

Comma-separated list of names of entry-point MATLAB functions that you want to analyze. Specify as character vectors or string scalars.

Examples

Identify Unsupported Functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, `foo1`, and the function `foo2` that `foo1` calls.

Write the function `foo2` and save it in the file `foo2.m`.

```
function out = foo2(in)
  out = eval(in);
end
```

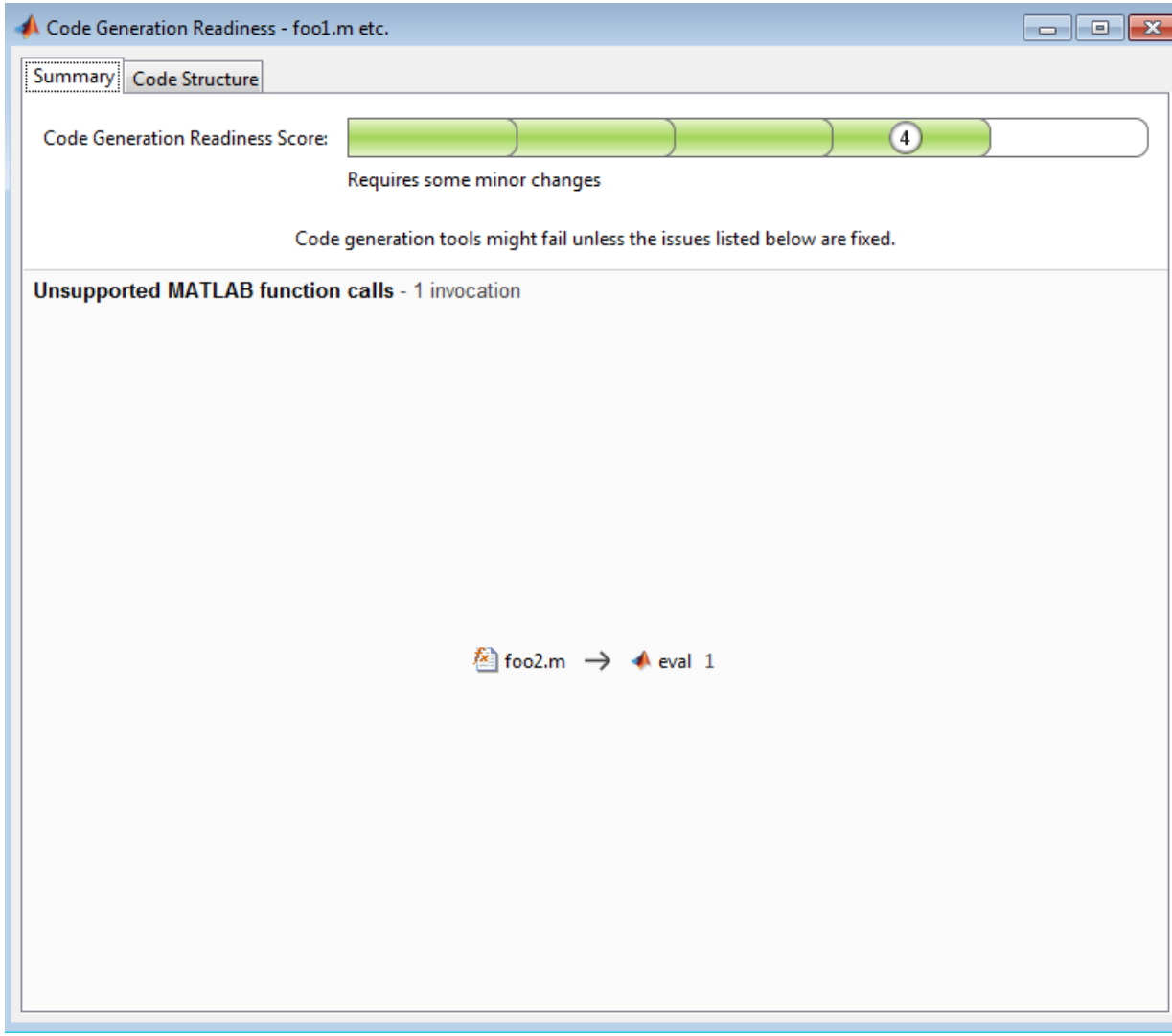
Write the function `foo1` that calls `foo2`. Save `foo1` in the file `foo1.m`.

```
function out = foo1(in)
  out = foo2(in);
  disp(out);
end
```

Analyze `foo1`.

```
coder.screener('foo1')
```

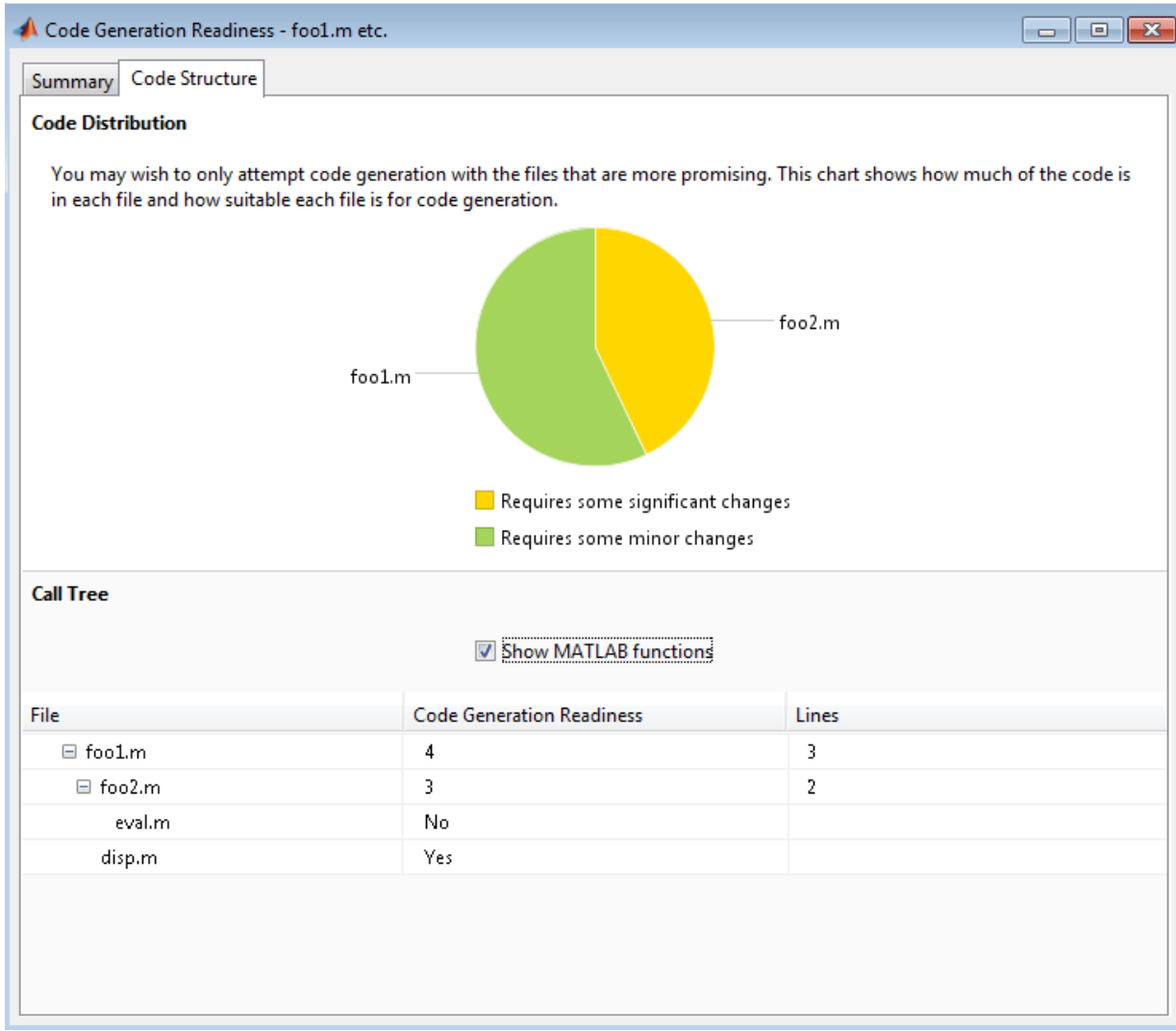
The code generation readiness report displays a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.



In the report, click the **Code Structure** tab and select the **Show MATLAB functions** check box.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo1.m` green to indicate that it is suitable for code generation.
- Colors `foo2.m` yellow to indicate that it requires significant changes.
- Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1–5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool does not detect issues.
- Displays a call tree.



The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function, which code generation does not support. To generate a MEX function for `foo2.m`, modify the code to make the call to `eval` extrinsic.

```
function out = foo2(in)
  coder.extrinsic('eval');
```

```
  out = eval(in);
end
```

Rerun the code generation readiness tool.

```
coder.screener('foo1')
```

The report no longer flags that code generation does not support the `eval` function. When you generate a MEX function for `foo1`, the code generator dispatches `eval` to MATLAB for execution. For standalone code generation, the code generator does not generate code for `eval`.

Identify Unsupported Data Types

The `coder.screener` function identifies MATLAB data types that code generation does not support.

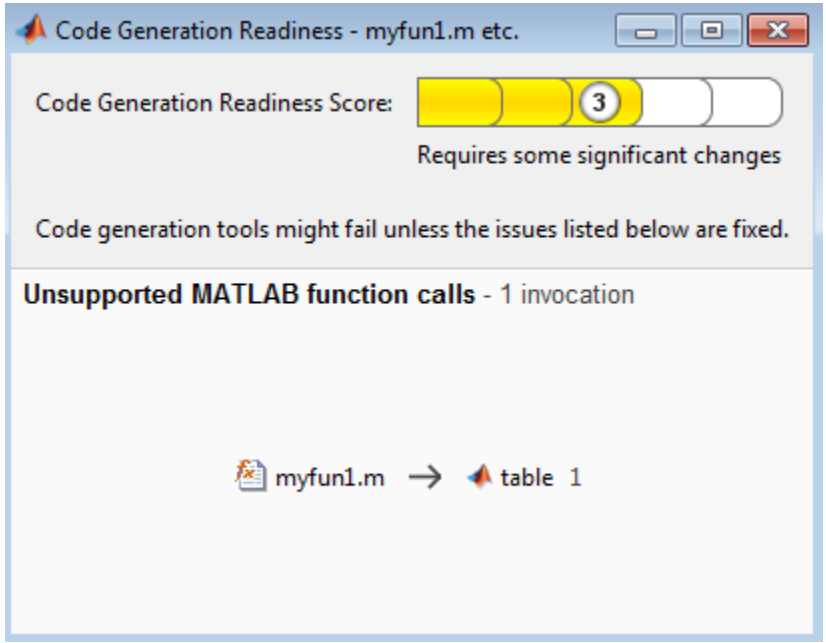
Write the function `myfun` that contains a MATLAB table.

```
function outTable = myfun1(A)
outTable = table(A);
end
```

Analyze `myfun`.

```
coder.screener('myfun1');
```

The code generation readiness report indicates that table data types are not supported for code generation.



The report assigns `myfun1` a code readiness score of `3`. Before generating code, you must fix the reported issues.

Tips

- Before using `coder.screener`, fix issues that the Code Analyzer identifies.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.
- It is possible that `coder.screener` does not detect all issues, and can report false errors. Therefore, before generating C code, verify that your code is suitable for code generation by generating a MEX function.

Alternatives

- “Run Code Generation Readiness Tool from the Current Folder Browser”

- “Run the Code Generation Readiness Tool Using the MATLAB Coder App”.

See Also

codegen

Topics

“MATLAB Language Features Supported for C/C++ Code Generation”
“Functions and Objects Supported for C/C++ Code Generation — Alphabetical List”
“Functions and Objects Supported for C/C++ Code Generation — Category List”
“Code Generation Readiness Tool”

Introduced in R2012b

coder.target

Determine if code generation target is specified target

Syntax

```
tf = coder.target(target)
```

Description

`tf = coder.target(target)` returns true (1) if the code generation target is `target`. Otherwise, it returns false (0).

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns true.

Examples

Use `coder.target` to Parametrize a MATLAB Function

Parametrize a MATLAB function so that it works in MATLAB or in generated code. When the function runs in MATLAB, it calls the MATLAB function `myabsval`. The generated code, however, calls a C library function `myabsval`.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)
%#codegen
y = abs(u);
```

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```
function y = callmyabsval(y)
%#codegen
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
  % Executing in MATLAB, call function myabsval
  y = myabsval(y);
else
  % add the required include statements to generated function code
  coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
  coder.cinclude('myabsval_initialize.h');
  coder.cinclude('myabsval.h');
  coder.cinclude('myabsval_terminate.h');

  % Executing in the generated code.
  % Call the initialize function before calling the
  % C function for the first time
  coder.ceval('myabsval_initialize');

  % Call the generated C library function myabsval
  y = coder.ceval('myabsval',y);

  % Call the terminate function after
  % calling the C function for the last time
  coder.ceval('myabsval_terminate');
end
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use `coder.updateBuildInfo` to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsva
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex(-2.75)
```

```
ans =

    2.7500
```

Call the MATLAB function `callmyabsval`.

```
callmyabsval(-2.75)
```

```
ans =

    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

Input Arguments

`target` — code generation target

`'MATLAB'` | `'MEX'` | `'Sfun'` | `'Rtw'` | `'HDL '` | `'Custom'`

Code generation target, specified as a character vector or a string scalar. Specify one of these targets.

`'MATLAB'`	Running in MATLAB (not generating code)
`'MEX'`	Generating a MEX function
`'Sfun'`	Simulating a Simulink model
`'Rtw'`	Generating a LIB, DLL, or EXE target
`'HDL'`	Generating an HDL target
`'Custom'`	Generating a custom target

Example: `tf = coder.target('MATLAB')`

Example: `tf = coder.target("MATLAB")`

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.cinclude` | `coder.updateBuildInfo`

Topics

“Call C/C++ Code from MATLAB Code”

Introduced in R2011a

coder.typeof

Package: coder

Create `coder.Type` object to represent the type of an entry-point function input

Syntax

```
t = coder.typeof(v)
t = coder.typeof(v, sz, variable_dims)
t = coder.typeof(t)
```

Description

`t = coder.typeof(v)` creates an object that derives from `coder.Type` to represent the type of `v` for code generation. Use `coder.typeof` to specify only input parameter types. For example, use it with the `codegen` function `-args` option or in a MATLAB Coder project when you are defining an input type by example. Do not use it in MATLAB code from which you intend to generate code.

`t = coder.typeof(v, sz, variable_dims)` returns a modified copy of `t = coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions specified by `variable_dims`.

- If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size.
- When `sz` is `[]`, the upper bounds of `v` do not change.
- If you do not specify the `variable_dims` input parameter, the bounded dimensions of the type are fixed.
- A scalar `variable_dims` applies to all dimensions. However, if `variable_dims` is `1`, the size of a singleton dimension remains fixed.
- When `v` is a cell array whose elements have the same classes, but different sizes, if you specify variable-size dimensions, `coder.typeof` creates a homogeneous cell array type. If the elements have different classes, `coder.typeof` reports an error.

`t = coder.typeof(t)`, where `t` is a `coder.Type` object, returns `t` itself.

Input Arguments

`sz`

Size vector specifying each dimension of type object.

`t`

`coder.Type` object

`v`

MATLAB expression that describes the set of values represented by this type.

`v` can be a MATLAB numeric, logical, char, enumeration, or fixed-point array. `v` can also be a cell array, structure, or value class that contains the previous types.

`variable_dims`

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

For a cell array, if the elements have different classes, you cannot specify variable-size dimensions.

Output Arguments

`t`

`coder.Type` object

Examples

Create a type for a simple fixed-size `5x6` matrix of doubles.

```
coder.typeof(ones(5, 6))
 % returns 5x6 double
coder.typeof(0, [5 6])
 % also returns 5x6 double
```

Create a type for a variable-size matrix of doubles.

```
coder.typeof(ones(3,3), [], 1)
% returns :3 x :3 double
% ':' indicates variable-size dimensions
```

Create a type for a structure with a variable-size field.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
% Returns
% coder.StructType
%    1x1 struct
%      a:  :3x:5 double
%      b:  3x3  double
% ':' indicates variable-size dimensions
```

Create a type for a homogeneous cell array with a variable-size field.

```
a = coder.typeof(0,[3 5],1);
b = magic(3);
coder.typeof({a b})
% Returns
% coder.CellType
%   1x2 homogeneous cell
%      base: :3x:5 double
% ':' indicates variable-size dimensions
```

Create a type for a heterogeneous cell array.

```
a = coder.typeof('a');
b = coder.typeof(1);
coder.typeof({a b})
% Returns
% coder.CellType
%   1x2 heterogeneous cell
%      f0: 1x1 char
%      f1: 1x1 double
```

Create a variable-size homogeneous cell array type from a cell array that has the same class but different sizes.

1 Create a type for a cell array that contains two character vectors with different sizes. The cell array type is heterogeneous.

```
coder.typeof({'aa', 'bbb'})
% Returns
% coder.CellType
%   1x2 heterogeneous cell
%      f0: 1x2 char
%      f1: 1x3 char
```

2 Create a type using the same cell array input. This time, specify that the cell array type has variable-size dimensions. The cell array type is homogeneous.

```
coder.typeof({'aa','bbb'},[1,10],[0,1])
% Returns
% coder.CellType
%   1x:10 homogeneous cell
%      base: 1x:3 char
```

Create a type for a matrix with fixed-size and variable-size dimensions.

```
coder.typeof(0, [2,3,4], [1 0 1]);
% Returns :2x3x:4 double
% ':' indicates variable-size dimensions

coder.typeof(10, [1 5], 1)
% returns double 1 x  :5
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size.

```
coder.typeof(10,[inf,3])
% returns double:inf x 3
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size with an upper bound of 3.

```
coder.typeof(10, [inf,3],[0 1])
% returns double :inf x :3
% ':' indicates variable-size dimensions
```

Convert a fixed-size matrix to a variable-size matrix.

```
 coder.typeof(ones(5,5), [], 1)
% returns double :5x:5
% ':' indicates variable-size dimensions
```

Create a nested structure (a structure as a field of another structure).

```
S = struct('a',double(0),'b',single(0))
SuperS.x = coder.typeof(S)
SuperS.y = single(0)
coder.typeof(SuperS)
% Returns
% coder.StructType
% SuperS:  1x1 struct
%   with fields
%      x:  1x1 struct
%         with fields
%             a: 1x1 double
%             b: 1x1 single
%      y:  1x1  single
```

Create a structure containing a variable-size array of structures as a field.

```
S = struct('a',double(0),'b',single(0))
SuperS.x = coder.typeof(S,[1 inf],[0 1])
SuperS.y = single(0)
coder.typeof(SuperS)
% Returns
% coder.StructType
% SuperS:  1x1 struct
%   with fields
%      x:  1x:inf struct
%         with fields
%             a: 1x1 double
%             b: 1x1 single
%      y:  1x1  single
% ':' indicates variable-size dimensions
```

Create a type for a value class object

1 Create this value class:

```
classdef mySquare
    properties
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
```

```
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end
```

2 Create an object of `mySquare`.

```
sq_obj = coder.typeof(mySquare(4))

sq_obj =

coder.ClassType
   1×1 mySquare
      side: 1×1 double
```

3 Create a type for an object that has the same properties as `sq_obj`.

```
t = coder.typeof(sq_obj)

t =

coder.ClassType
   1×1 mySquare
      side: 1×1 double
```

Alternatively, you can create the type from the class definition:

```
t = coder.typeof(mySquare(4))

t =

coder.ClassType
   1×1 mySquare
      side: 1×1 double
```

Create a type for a string scalar

1 Define a string scalar. For example:

```
s = "mystring";
```

2 Create a type from `s`.

```
t = coder.typeof(s);
```

3. To make `t` variable-size, assign the `Value` property of `t` to a type for a variable-size character vector that has the upper bound that you want. For example, specify that type `t` is variable-size with an upper bound of 10.

   ```
   t.Properties.Value = coder.typeof('a',[1 10], [0 1]);
   ```

   To specify that `t` is variable-size with no upper bound:

   ```
   t.Properties.Value = coder.typeof('a',[1 inf]);
   ```

4. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Limitations

- For sparse matrices, `coder.typeof` drops upper bounds for variable-size dimensions.

Tips

- `coder.typeof` fixes the size of a singleton dimension unless the `variable_dims` argument explicitly specifies that the singleton dimension has a variable size.

  For example, the following code specifies a 1-by-:10 double. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

  ```
  t = coder.typeof(5,[1 10],1)
  ```

  By contrast, the following code specifies a :1-by-:10 double. Both dimensions have a variable size.

  ```
  t = coder.typeof(5,[1 10],[1 1])
  ```

  Note For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

- If you are already specifying the type of an input variable using a type function, do not use `coder.typeof` unless you also want to specify the size. For instance, instead of `coder.typeof(single(0))`, use the syntax `single(0)`.
- For cell array types, `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and

size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type where the first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the coder.CellType `makeHomogeneous` or `makeHeterogeneous` methods to make a type with the classification that you want. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as heterogeneous and homogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

See Also

`codegen` | `coder.ArrayType` | `coder.CellType` | `coder.ClassType` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` | `coder.resize`

Topics

“Define Input Properties by Example at the Command Line”
“Specify Cell Array Inputs at the Command Line”
“Specify Objects as Inputs at the Command Line”
“Define String Scalar Inputs”

Introduced in R2011a

coder.unroll

Unroll `for`-loop by making a copy of the loop body for each loop iteration

Syntax

```
coder.unroll()
coder.unroll(flag)
```

Description

`coder.unroll()` unrolls a `for`-loop. The `coder.unroll` call must be on a line by itself immediately preceding the `for`-loop that it unrolls.

Instead of producing a `for`-loop in the generated code, loop unrolling produces a copy of the `for`-loop body for each loop iteration. In each iteration, the loop index becomes constant. To unroll a loop, the code generator must be able to determine the bounds of the `for`-loop.

For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can increase code generation time significantly and generate inefficient code.

`coder.unroll` is ignored outside of code generation.

`coder.unroll(flag)` unrolls a `for`-loop if `flag` is `true`. `flag` is evaluated at code generation time. The `coder.unroll` call must be on a line by itself immediately preceding the `for`-loop that it unrolls.

Examples

Unroll a `for`-loop

To produce copies of a `for`-loop body in the generated code, use `coder.unroll`.

In one file, write the entry-point function `call_getrand` and a local function `getrand`. `getrand` unrolls a `for`-loop that assigns random numbers to an n-by-1 array. `call_getrand` calls `getrand` with the value 3.

```
function z = call_getrand
%#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
coder.unroll();
for i = 1:n
    y(i) = rand();
end
end
```

Generate a static library.

```
codegen -config:lib call_getrand -report
```

In the generated code, the code generator produces a copy of the `for`-loop body for each of the three loop iterations.

```
static void getrand(double y[3])
{
  y[0] = b_rand();
  y[1] = b_rand();
  y[2] = b_rand();
}
```

Control `for`-loop Unrolling with Flag

Control loop unrolling by using `coder.unroll` with the `flag` argument.

In one file, write the entry-point function `call_getrand_unrollflag` and a local function `getrand_unrollflag`. When the number of loop iterations is less than 10, `getrand_unrollflag` unrolls the `for`-loop. `call_getrand` calls `getrand` with the value 50.

```
function z = call_getrand_unrollflag
%#codegen
z = getrand_unrollflag(50);
```

```
end

function y = getrand_unrollflag(n)
coder.inline('never');
unrollflag = n < 10;
y = zeros(n, 1);
coder.unroll(unrollflag)
for i = 1:n
    y(i) = rand();
end
end
```

Generate a static library.

```
codegen -config:lib call_getrand_unrollflag -report
```

The number of iterations is not less than 10. Therefore, the code generator does not unroll the `for`-loop. It produces a `for`-loop in the generated code.

```
static void getrand_unrollflag(double y[50])
{
  int i;
  for (i = 0; i < 50; i++) {
    y[i] = b_rand();
  }
}
```

Use Legacy Syntax to Unroll `for`-Loop

```
function z = call_getrand
%#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
for i = coder.unroll(1:n)
    y(i) = rand();
end
end
```

Use Legacy Syntax to Control `for`-Loop Unrolling

```
function z = call_getrand_unrollflag
%#codegen
z = getrand_unrollflag(50);
end

function y = getrand_unrollflag(n)
coder.inline('never');
unrollflag = n < 10;
y = zeros(n, 1);
for i = coder.unroll(1:n, unrollflag)
    y(i) = rand();
end
end
```

Input Arguments

`flag` — Indicates whether to unroll the `for`-loop
true (default) | false

When `flag` is `true`, the code generator unrolls the `for`-loop. When `flag` is `false`, the code generator produces a `for`-loop in the generated code. `flag` is evaluated at code generation time.

Tips

- Sometimes, the code generator unrolls a `for`-loop even though you do not use `coder.unroll`. For example, if a `for`-loop indexes into a heterogeneous cell array or into `varargin` or `varargout`, the code generator unrolls the loop. By unrolling the loop, the code generator can determine the value of the index for each loop iteration. The code generator uses heuristics to determine when to unroll a `for`-loop. If the heuristics fail to identify that unrolling is warranted, or if the number of loop iterations exceeds a limit, code generation fails. In these cases, you can force loop unrolling by using `coder.unroll`. See “Nonconstant Index into varargin or varargout in a for-Loop”.
- If a `for`-loop is not preceded by `coder.unroll`, the code generator uses a loop unrolling threshold to determine whether to automatically unroll the loop. If the number of loop iterations is less than the threshold, the code generator unrolls the

loop. If the number of iterations is greater than or equal to the threshold, the code generator produces a `for`-loop. The default value of the threshold is `5`. By modifying this threshold, you can fine-tune loop unrolling. For more details, see “Unroll for-Loops”.

See Also

`coder.inline`

Topics

“Unroll for-Loops”
“Nonconstant Index into varargin or varargout in a for-Loop”

Introduced in R2011a

coder.updateBuildInfo

Update build information object `RTW.BuildInfo`

Syntax

```
coder.updateBuildInfo('addCompileFlags',options)
coder.updateBuildInfo('addLinkFlags',options)
coder.updateBuildInfo('addDefines',options)
coder.updateBuildInfo( ___ ,group)

coder.updateBuildInfo('addLinkObjects',filename,path)
coder.updateBuildInfo('addLinkObjects',filename,path,priority,
precompiled)
coder.updateBuildInfo('addLinkObjects',filename,path,priority,
precompiled,linkonly)
coder.updateBuildInfo( ___ ,group)

coder.updateBuildInfo('addNonBuildFiles',filename)
coder.updateBuildInfo('addSourceFiles',filename)
coder.updateBuildInfo('addIncludeFiles',filename)
coder.updateBuildInfo( ___ ,path)
coder.updateBuildInfo( ___ ,path,group)

coder.updateBuildInfo('addSourcePaths',path)
coder.updateBuildInfo('addIncludePaths',path)
coder.updateBuildInfo( ___ ,group)
```

Description

`coder.updateBuildInfo('addCompileFlags',options)` adds compiler options to the build information object.

`coder.updateBuildInfo('addLinkFlags',options)` adds link options to the build information object.

`coder.updateBuildInfo('addDefines',options)` adds preprocessor macro definitions to the build information object.

`coder.updateBuildInfo(___,group)` assigns a group name to `options` for later reference.

`coder.updateBuildInfo('addLinkObjects',filename,path)` adds a link object from a file to the build information object.

`coder.updateBuildInfo('addLinkObjects',filename,path,priority,precompiled)` specifies if the link object is precompiled.

`coder.updateBuildInfo('addLinkObjects',filename,path,priority,precompiled,linkonly)` specifies if the object is to be built before being linked or used for linking alone. If the object is to be built, it specifies if the object is precompiled.

`coder.updateBuildInfo(___,group)` assigns a group name to the link object for later reference.

`coder.updateBuildInfo('addNonBuildFiles',filename)` adds a nonbuild-related file to the build information object.

`coder.updateBuildInfo('addSourceFiles',filename)` adds a source file to the build information object.

`coder.updateBuildInfo('addIncludeFiles',filename)` adds an include file to the build information object.

`coder.updateBuildInfo(___,path)` adds the file from specified path.

`coder.updateBuildInfo(___,path,group)` assigns a group name to the file for later reference.

`coder.updateBuildInfo('addSourcePaths',path)` adds a source file path to the build information object.

`coder.updateBuildInfo('addIncludePaths',path)` adds an include file path to the build information object.

`coder.updateBuildInfo(___,group)` assigns a group name to the path for later reference.

Examples

Add Multiple Compiler Options

Add the compiler options `-Zi` and `-Wall` during code generation for function, `func`.

Anywhere in the MATLAB code for `func`, add the following line:

```
coder.updateBuildInfo('addCompileFlags','-Zi -Wall');
```

Generate code for `func` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport func
```

You can see the added compiler options under the **Build Logs** tab in the Code Generation Report.

Add Source File Name

Add a source file to the project build information while generating code for a function, `calc_factorial`.

1. Write a header file `fact.h` that declares a C function `factorial`.

   ```
   double factorial(double x);
   ```

   `fact.h` will be included as a header file in generated code. This inclusion ensures that the function is declared before it is called.

   Save the file in the current folder.

2. Write a C file `fact.c` that contains the definition of `factorial`. `factorial` calculates the factorial of its input.

   ```
   #include "fact.h"

         double factorial(double x)
         {
             int i;
             double fact = 1.0;
             if (x == 0 || x == 1) {
   ```

```
            return 1.0;
        } else {
            for (i = 1; i <= x; i++) {
                fact *= (double)i;
            }
            return fact;
        }
    }
```

`fact.c` is used as a source file during code generation.

Save the file in the current folder.

3 Write a MATLAB function `calc_factorial` that uses `coder.ceval` to call the external C function `factorial`.

Use `coder.updateBuildInfo` with option `'addSourceFiles'` to add the source file `fact.c` to the build information. Use `coder.cinclude` to include the header file `fact.h` in the generated code.

```
function y = calc_factorial(x) %#codegen

  coder.cinclude('fact.h');
  coder.updateBuildInfo('addSourceFiles', 'fact.c');

  y = 0;
  y = coder.ceval('factorial', x);
```

4 Generate code for `calc_factorial` using the `codegen` command.

```
codegen -config:dll -launchreport calc_factorial -args 0
```

In the Code Generation Report, on the **C Code** tab, you can see the added source file `fact.c`.

Add Link Object

Add a link object `LinkObj.lib` to the build information while generating code for a function `func`. For this example, you must have a link object `LinkObj.lib` saved in a local folder, for example, `c:\Link_Objects`.

Anywhere in the MATLAB code for `func`, add the following lines:

```
libPriority = '';
libPreCompiled = true;
libLinkOnly = true;
libName = 'LinkObj.lib';
libPath = 'c:\Link_Objects';
coder.updateBuildInfo('addLinkObjects', libName, libPath, ...
     libPriority, libPreCompiled, libLinkOnly);
```

Generate a MEX function for `func` using the `codegen` command. Open the Code Generation Report.

```
codegen -launchreport func
```

You can see the added link object under the **Build Logs** tab in the Code Generation Report.

Add Include Paths

Add an include path to the build information while generating code for a function, `adder`. Include a header file, `adder.h`, existing on the path.

When header files do not reside in the current folder, to include them, use this method:

1. Write a header file `mysum.h` that contains the declaration for a C function `mysum`.

   ```
   double mysum(double, double);
   ```

   Save it in a local folder, for example `c:\coder\myheaders`.

2. Write a C file `mysum.c` that contains the definition of the function `mysum`.

   ```
   #include "mysum.h"

   double mysum(double x, double y)
    {
     return(x+y);
    }
   ```

   Save it in the current folder.

3. Write a MATLAB function `adder` that adds the path `c:\coder\myheaders` to the build information.

Use `coder.cinclude` to include the header file `mysum.h` in the generated code.

```
function y = adder(x1, x2) %#codegen
   coder.updateBuildInfo('addIncludePaths','c:\coder\myheaders');
   coder.updateBuildInfo('addSourceFiles','mysum.c');
     %Include the source file containing C function definition
   coder.cinclude('mysum.h');
   y = 0;
   if coder.target('MATLAB')
      % This line ensures that the function works in MATLAB
        y = x1 + x2;
   else
       y = coder.ceval('mysum', x1, x2);
    end
end
```

4 Generate code for `adder` using the `codegen` command.

```
codegen -config:lib -launchreport adder -args {0,0}
```

Open the Code Generation Report. The header file `adder.h` is included in the generated code.

Input Arguments

`options` — Build options

character vector | string scalar

Build options, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, `options` specifies the relevant build options to be added to the project's build information.

Leading Argument	Values in `options`
`'addCompileFlags'`	Compiler options
`'addLinkFlags'`	Link options
`'addDefines'`	Preprocessor macro definitions

The function adds the options to the end of an option vector.

Example: `coder.updateBuildInfo('addCompileFlags','-Zi -Wall')`

`group` — Group name

character vector | string scalar

Name of user-defined group, specified as a character vector or string scalar. The value must be a compile-time constant.

The `group` option assigns a group name to the parameters in the second argument.

Leading Argument	Second Argument	Parameters Named by `group`
`'addCompileFlags'`	`options`	Compiler options
`'addLinkFlags'`	`options`	Link options
`'addLinkObjects'`	`filename`	Name of file containing linkable objects
`'addNonBuildFiles'`	`filename`	Name of nonbuild-related file
`'addSourceFiles'`	`filename`	Name of source file
`'addSourcePaths'`	`path`	Name of source file path

You can use `group` to:

- Document the use of specific parameters.
- Retrieve or apply multiple parameters together as one group.

`filename` — File name

character vector | string scalar

File name, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, `filename` specifies the relevant file to be added to the project's build information.

Leading Argument	File Specified by `filename`
`'addLinkObjects'`	File containing linkable objects
`'addNonBuildFiles'`	Nonbuild-related file
`'addSourceFiles'`	Source file

The function adds the file name to the end of a file name vector.

Example: `coder.updateBuildInfo('addSourceFiles', 'fact.c')`

`path` — Path name

character vector | string scalar

Relative path name, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, `path` specifies the relevant path name to be added to the project's build information. The function adds the path to the end of a path name vector.

Leading Argument	Path Specified by `path`
`'addLinkObjects'`	Path to linkable objects
`'addNonBuildFiles'`	Path to nonbuild-related files
`'addSourceFiles'`, `'addSourcePaths'`	Path to source files

The relative path starts from the *build folder*. If you have a function `foo` contained in the folder `C:\myCode`, and you generate MEX code by using:

```
codegen foo -report
```

then the build folder is `C:\myCode\codegen\mex\foo`. You can write the path from the build folder or write the path from the current working folder in which you generate code. Reference the current working folder by using the `START_DIR` macro. For example, suppose that your source file is contained in `C:\myCode\mySrcDir`, and you generate code from `C:\myCode`. Write the path as in these examples:

Example: `coder.updateBuildInfo('addSourceFiles','fact.c','..\..\..\mySrcDir')`

Example: `coder.updateBuildInfo('addSourceFiles','fact.c','$(START_DIR)\mySrcDir')`

`priority` — Relative priority of link object

`''`

Priority of link objects.

This feature applies only when several link objects are added. Currently, only a single link object file can be added for every `coder.updateBuildInfo` statement. Therefore, this feature is not available for use.

To use the succeeding arguments, include `''` as a placeholder argument.

`precompiled` — Variable indicating if link objects are precompiled
logical value

Variable indicating if the link objects are precompiled, specified as a logical value. The value must be a compile-time constant.

If the link object has been prebuilt for faster compiling and linking and exists in a specified location, specify `true`. Otherwise, the MATLAB Coder build process creates the link object in the build folder.

If `linkonly` is set to `true`, this argument is ignored.

Data Types: `logical`

`linkonly` — Variable indicating if objects must be used for linking only
logical value

Variable indicating if objects must be used for linking only, specified as a logical value. The value must be a compile-time constant.

If you want that the MATLAB Coder build process must not build or generate rules in the makefile for building the specified link object, specify `true`. Instead, when linking the final executable, the process should just include the object. Otherwise, rules for building the link object are added to the makefile.

You can use this argument to incorporate link objects for which source files are not available.

If `linkonly` is set to `true`, the value of `precompiled` is ignored.

Data Types: `logical`

See Also

`codegen` | `coder.ExternalDependency` | `coder.ceval` | `coder.cinclude` | `coder.ref` | `coder.rref` | `coder.target` | `coder.wref`

Topics

“Build Process Customization”
“Call C/C++ Code from MATLAB Code”
“Configure Build for External C/C++ Code”
“Configure Build Settings”

Introduced in R2013b

coder.varsize

Package: coder

Declare variable-size data

Syntax

```
coder.varsize(varName1,...,varNameN)
coder.varsize(varName1,...,varNameN,ubounds)
coder.varsize(varName1,...,varNameN,ubounds,dims)
```

Description

`coder.varsize(varName1,...,varNameN)` declares that the variables named `varName1,...,varNameN` have a variable size. The declaration instructs the code generator to allow the variables to change size during execution of the generated code. With this syntax, you do not specify the upper bounds of the dimensions of the variables or which dimensions can change size. The code generator computes the upper bounds. All dimensions, except singleton dimensions on page 2-212, are allowed to change size.

Use `coder.varsize` according to these restrictions and guidelines:

- Use `coder.varsize` inside a MATLAB function intended for code generation.
- The `coder.varsize` declaration must precede the first use of a variable. For example:

  ```
  ...
  x = 1;
  coder.varsize('x');
  disp(size(x));
  ...
  ```
- Use `coder.varsize` to declare that an output argument has a variable size or to address size mismatch errors. Otherwise, to define variable-size data, use the methods described in “Define Variable-Size Data for Code Generation”.

Note For MATLAB Function blocks, to declare variable-size input or output signals, use the Ports and Data Manager. See “Declare Variable-Size Inputs and Outputs” (Simulink). If you provide upper bounds in a `coder.varsize` declaration, the upper bounds must match the upper bounds in the Ports and Data Manager.

For more restrictions and guidelines, see “Limitations” on page 2-210 and “Tips” on page 2-212.

`coder.varsize(varName1,...,varNameN,ubounds)` also specifies an upper bound for each dimension of the variables. All variables must have the same number of dimensions. All dimensions, except singleton dimensions on page 2-212, are allowed to change size.

`coder.varsize(varName1,...,varNameN,ubounds,dims)` also specifies an upper bound for each dimension of the variables and whether each dimension has a fixed size or a variable size. If a dimension has a fixed size, then the corresponding `ubound` element specifies the fixed size of the dimension. All variables have the same fixed-size dimensions and the same variable-size dimensions.

Examples

Address Size Mismatch Error by Using `coder.varsize`

After a variable is used (read), changing the size of the variable can cause a size mismatch error. Use `coder.varsize` to specify that the size of the variable can change.

Code generation for the following function produces a size mismatch error because `x = 1:10` changes the size of the second dimension of `x` after the line `y = size(x)` that uses `x`.

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
y = size(x);
if n > 10
    x = 1:10;
end
```

To declare that `x` can change size, use `coder.varsize`.

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
coder.varsize('x');
y = size(x);
if n > 10
    x = 1:10;
end
```

If you remove the line `y = size(x)`, you no longer need the `coder.varsize` declaration because `x` is not used before its size changes.

Declare Variable-Size Array with Upper Bounds

Specify that `A` is a row vector whose second dimension has a variable size with an upper bound of 20.

```
function fcn()
...
coder.varsize('A',[1 20]);
...
end
```

When you do not provide `dims`, all dimensions, except singleton dimensions, have a variable size.

Declare Variable-Size Array with a Mix of Fixed and Variable Dimensions

Specify that `A` is an array whose first dimension has a fixed size of three and whose second dimension has a variable size with an upper bound of 20.

```
function fcn()
...
coder.varsize('A',[3 20], [0 1] );
...
end
```

Declare Variable-Size Structure Fields

If a structure field belongs to an array of structures, use colon (`:`) as the index expression to make the field variable-size for all elements of the array.

In this function, the statement `coder.varsize('data(:).values')` declares that the field `values` inside each element of `data` has a variable size.

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

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

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

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

Declare Variable-Size Cell Array

Specify that cell array `C` has a fixed-size first dimension and variable-size second dimension with an upper bound of three. The `coder.varsize` declaration must precede the first use of `C`.

```
...
C = {1 [1 2]};
coder.varsize('C', [1 3], [0 1]);
y = C{1};
...
end
```

Without the `coder.varsize` declaration, `C` is a heterogeneous cell array whose elements have the same class and different sizes. With the `coder.varsize` declaration, `C` is a

homogeneous cell array whose elements have the same class and maximum size. The first dimension of each element is fixed at 1. The second dimension of each element has a variable size with an upper bound of 2.

Declare That a Cell Array Has Variable-Size Elements

Specify that the elements of cell array `C` are vectors with a fixed-size first dimension and variable-size second dimension with an upper bound of 5.

```
...
C = {1 2 3};
coder.varsize('C{:}', [1 5], [0 1]);
C = {1, 1:5, 2:3};
...
```

Input Arguments

`varName1,...,varNameN` — Names of variables to declare as having a variable size

character vectors | string scalars

Names of variables to declare as having a variable size, specified as one or more character vectors or string scalars.

Example: `coder.varsize('x','y')`

`ubounds` — Upper bounds for array dimensions

[] (default) | vector of integer constants

Upper bounds for array dimensions, specified as a vector of integer constants.

When you do not specify `ubounds`, the code generator computes the upper bound for each variable. If the `ubounds` element corresponds to a fixed-size dimension, the value is the fixed size of the dimension.

Example: `coder.varsize('x','y',[1 2])`

`dims` — Indication of whether each dimension has a fixed size or a variable size

logical vector

Indication of whether each dimension has a fixed size or a variable size, specified as a logical vector. Dimensions that correspond to 0 or `false` in `dims` have a fixed size. Dimensions that correspond to 1 or `true` have a variable size.

When you do not specify `dims`, the dimensions have a variable size, except for the singleton dimensions.

Example: `coder.varsize('x','y',[1 2], [0 1])`

Limitations

- The `coder.varsize` declaration instructs the code generator to allow the size of a variable to change. It does not change the size of the variable. Consider this code:

  ```
  ...
  x = 7;
  coder.varsize('x', 1,5]);
  disp(size(x));
  ...
  ```

  After the `coder.varsize` declaration, `x` is still a 1-by-1 array. You cannot assign a value to an element beyond the current size of `x`. For example, this code produces a run-time error because the index 3 exceeds the dimensions of `x`.

  ```
  ...
  x = 7;
  coder.varsize('x', [1,5]);
  x(3) = 1;
  ...
  ```

- `coder.varsize` is not supported for a function input argument. Instead:
  - If the function is an entry-point function, specify that an input argument has a variable size by using `coder.typeof` at the command line. Alternatively, specify that an entry-point function input argument has a variable size by using the **Define Input Types** step of the app.
  - If the function is not an entry-point function, use `coder.varsize` in the calling function with the variable that is the input to the called function.
- For sparse matrices, `coder.varsize` drops upper bounds for variable-size dimensions.
- Limitations for using `coder.varsize` with cell arrays:

- A cell array can have a variable size only if it is homogeneous. When you use `coder.varsize` with a heterogeneous cell array, the code generator tries to make the cell array homogeneous. The code generator tries to find a class and maximum size that apply to all elements of the cell array. For example, consider the cell array `c = {1, [2 3]}`. Both elements can be represented by a double type whose first dimension has a fixed size of 1 and whose second dimension has a variable size with an upper bound of 2. If the code generator cannot find a common class and a maximum size, code generation fails. For example, consider the cell array `c = {'a',[2 3]}`. The code generator cannot find a class that can represent both elements because the first element is `char` and the second element is `double`.
- If you use the `cell` function to define a fixed-size cell array, you cannot use `coder.varsize` to specify that the cell array has a variable size. For example, this code causes a code generation error because `x = cell(1,3)` makes `x` a fixed-size,1-by-3 cell array.

  ```
  ...
  x = cell(1,3);
  coder.varsize('x',[1 5])
  ...
  ```

  You can use `coder.varsize` with a cell array that you define by using curly braces. For example:

  ```
  ...
  x = {1 2 3};
  coder.varsize('x',[1 5])
  ...
  ```
- To create a variable-size cell array by using the `cell` function, use this code pattern:

  ```
  function mycell(n)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  end
  end
  ```

  See “Definition of Variable-Size Cell Array by Using cell”.

  To specify upper bounds for the cell array, use `coder.varsize`.

  ```
  function mycell(n)
  %#codegen
  ```

```
    x = cell(1,n);
    for i = 1:n
        x{i} = i;
    coder.varsize('x',[1,20]);
    end
    end
```

- `coder.varsize` is not supported for:
  - Global variables
  - MATLAB classes or class properties
  - String scalars

Definitions

Singleton Dimension

Dimension for which `size(A,dim) = 1`.

Tips

- In a code generation report or a MATLAB Function report, a colon (:) indicates that a dimension has a variable size. For example, a size of `1x:2` indicates that the first dimension has a fixed size of one and the second dimension has a variable size with an upper bound of two.
- If you use `coder.varsize` to specify that the upper bound of a dimension is 1, by default, the dimension has a fixed size of 1. To specify that the dimension can be 0 (empty array) or 1, set the corresponding element of the `dims` argument to `true`. For example, this code specifies that the first dimension of `x` has a fixed size of 1 and the other dimensions have a variable size of 5.

  ```
  coder.varsize('x',[1,5,5])
  ```

  In contrast, this code specifies that the first dimension of `x` has an upper bound of 1 and has a variable size (can be 0 or 1).

  ```
  coder.varsize('x',[1,5,5],[1,1,1])
  ```

Note For a MATLAB Function block, you cannot specify that an input or output signal with size 1 has a variable size.

- If you use input variables or the result of a computation using input variables to specify the size of an array, it is declared as variable-size in the generated code. Do not re-use `coder.varsize` on the array, unless you also want to specify an upper bound for its size.
- If you do not specify upper bounds with a `coder.varsize` declaration and the code generator is unable to determine the upper bounds, the generated code uses dynamic memory allocation. Dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, specify the upper bounds by providing the `ubounds` argument.

See Also

`coder.typeof`

Topics

“Code Generation for Variable-Size Arrays”
“Incompatibilities with MATLAB in Variable-Size Support for Code Generation”
“Avoid Duplicate Functions in Generated Code”

Introduced in R2011a

coder.wref

Indicate write-only data to pass by reference

Syntax

```
coder.wref(arg)
```

Description

`coder.wref(arg)` indicates that `arg` is a write-only expression or variable to pass by reference to an external C/C++ function. Use `coder.wref` only inside a `coder.ceval` call. This function enables the code generator to optimize the generated code by ignoring prior assignments to `arg` in your MATLAB code, because the external function is assumed to not read from the data. Write to all the elements of `arg` in your external code to fully initialize the memory.

Note The C/C++ function must fully initialize the memory referenced by `coder.wref(arg)`. Initialize the memory by assigning values to every element of `arg` in your C/C++ code. If the generated code tries to read from uninitialized memory, it can cause undefined run-time behavior.

See also `coder.ref` and `coder.rref`.

Examples

Pass Array by Reference as Write-Only

Suppose that you have a C function `init_array`.

```
void init_array(double* array, int numel) {
  for(int i = 0; i < numel; i++) {
    array[i] = 42;
```

```
  }
}
```

The C function defines the input variable `array` as a pointer to a double.

Call the C function `init_array` to initialize all elements of `y` to 42:

```
...
Y = zeros(5, 10);
coder.ceval('init_array', coder.wref(Y), int32(numel(Y)));
...
```

Pass Multiple Arguments as a Write-Only Reference

```
...
U = zeros(5, 10);
V = zeros(5, 10);
coder.ceval('my_fcn', coder.wref(U), int32(numel(U)), coder.wref(V), int32(numel(V)));
...
```

Pass Class Property as a Write-Only Reference

```
...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.wref(x.prop));
...
```

Pass Structure as a Write-Only Reference

To indicate that the structure type is defined in a C header file, use `coder.cstructname`.

Suppose that you have the C function `init_struct`. This function writes to the input argument but does not read from it.

```
#include "MyStruct.h"

void init_struct(struct MyStruct *my_struct)
{
   my_struct->f1 = 1;
   my_struct->f2 = 2;
}
```

The C header file, `MyStruct.h`, defines a structure type named `MyStruct`:

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

void init_struct(struct MyStruct *my_struct);

#endif
```

In your MATLAB function, pass a structure as a write-only reference to `init_struct`. Use `coder.cstructname` to indicate that the structure type for `s` has the name `MyStruct` that is defined in the C header file `MyStruct.h`.

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','init_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
coder.ceval('init_struct', coder.wref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

Pass Structure Field as a Write-Only Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.wref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
```

```
a = struct('b',b);
coder.ceval('foo', coder.wref(a.b(3,4).c(2).u));
...
```

Input Arguments

`arg` — Argument to pass by reference

scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`
Complex Number Support: Yes

Limitations

- You cannot pass these data types by reference:
  - Class or System object
  - Cell array or index into a cell array
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- If `arg` is an array, then `coder.wref(arg)` provides the address of the first element of the array. The `coder.wref(arg)` function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  ```
  coder.ceval('myFun',coder.wref(arg),int32(numel(arg));
  ```

- When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.

- In MATLAB, `coder.wref` results in an error. To parametrize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
- You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.rref`

Topics

"Call C/C++ Code from MATLAB Code"

Introduced in R2011a

parfor

Parallel `for`-loop

Syntax

```
parfor LoopVar = InitVal:EndVal; Statements; end
parfor (LoopVar = InitVal:EndVal, NumThreads); Statements; end
```

Description

`parfor LoopVar = InitVal:EndVal; Statements; end` creates a loop in a generated MEX function or in C/C++ code that runs in parallel on shared-memory multicore platforms.

The `parfor`-loop executes the `Statements` for values of `LoopVar` between `InitVal` and `Endval`. `LoopVar` specifies a vector of integer values increasing by 1.

`parfor (LoopVar = InitVal:EndVal, NumThreads); Statements; end` uses a maximum of `NumThreads` threads when creating a parallel `for`-loop.

Examples

Generate MEX for `parfor`

Generate a MEX function for a `parfor`-loop to execute on the maximum number of cores available.

Write a MATLAB function, `test_parfor`, that calls the fast Fourier transform function, `fft`, in a `parfor`-loop. Because the loop iterations run in parallel, this evaluation can be completed much faster than an analogous `for`-loop.

```
function a = test_parfor %#codegen
a = ones(10,256);
r = rand(10,256);
```

```
parfor i = 1:10
    a(i,:) = real(fft(r(i)));
end
```

Generate a MEX function for `test_parfor`. At the MATLAB command line, enter:

```
codegen test_parfor
```

`codegen` generates a MEX function, `test_parfor_mex`, in the current folder.

Run the MEX function. At the MATLAB command line, enter:

```
test_parfor_mex
```

The MEX function runs using the available cores.

Specify Maximum Number of Threads for `parfor`

Specify the maximum number of threads when generating a MEX function for a `parfor`-loop.

Write a MATLAB function, `specify_num_threads`, that uses input, `u`, to specify the maximum number of threads in the `parfor`-loop.

```
function y = specify_num_threads(u) %#codegen
  y = ones(1,100);
  % u specifies maximum number of threads
  parfor (i = 1:100,u)
    y(i) = i;
  end
end
```

Generate a MEX function for `specify_num_threads`. Use `-args 0` to specify the type of the input. At the MATLAB command line, enter:

```
% -args 0 specifies that input u is a scalar double
% u is typecast to an integer by the code generator
codegen -report specify_num_threads -args 0
```

`codegen` generates a MEX function, `specify_num_threads_mex`, in the current folder.

Run the MEX function, specifying that it run in parallel on at most four threads. At the MATLAB command line, enter:

```
specify_num_threads_mex(4)
```

The generated MEX function runs on up to four cores. If fewer than four cores are available, the MEX function runs on the maximum number of cores available at the time of the call.

Generate MEX for `parfor` Without Parallelization

Disable parallelization before generating a MEX function for a `parfor`-loop.

Write a MATLAB function, `test_parfor`, that calls the fast Fourier transform function, `fft`, in a `parfor`-loop.

```
function a = test_parfor %#codegen
a = ones(10,256);
r = rand(10,256);
parfor i = 1:10
    a(i,:) = real(fft(r(i)));
end
```

Generate a MEX function for `test_parfor`. Disable the use of OpenMP so that `codegen` does not generate a MEX function that can run on multiple threads.

```
codegen -O disable:OpenMP test_parfor
```

`codegen` generates a MEX function, `test_parfor_mex`, in the current folder.

Run the MEX function.

```
test_parfor_mex
```

The MEX function runs on a single thread.

If you disable parallelization, MATLAB Coder treats `parfor`-loops as `for`-loops. The software generates a MEX function that runs on a single thread. Disable parallelization to

compare performance of the serial and parallel versions of the generated MEX function or C/C++ code. You can also disable parallelization to debug issues with the parallel version.

Input Arguments

`LoopVar` — Loop index

integer

Loop index variable whose initial value is `InitVal` and final value is `EndVal`.

`InitVal` — Initial value of loop index

integer

Initial value for loop index variable, `Loopvar`. With `EndVal`, specifies the `parfor` range vector, which must be of the form `M:N`.

`EndVal` — Final value of loop index

integer

Final value for loop index variable, `LoopVar`. With `InitVal`, specifies the `parfor` range vector, which must be of the form `M:N`.

`Statements` — Loop body

text

The series of MATLAB commands to execute in the `parfor`-loop.

If you put more than one statement on the same line, separate the statements with semicolons. For example:

```
parfor i = 1:10
 arr(i) = rand(); arr(i) = 2*arr(i)-1;
end
```

`NumThreads` — Maximum number of threads running in parallel

number of available cores (default) | nonnegative integer

Maximum number of threads to use. If you specify the upper limit, MATLAB Coder uses no more than this number, even if additional cores are available. If you request more threads than the number of available cores, MATLAB Coder uses the maximum number of cores available at the time of the call. If the loop iterations are fewer than the threads, some threads perform no work.

If the parfor-loop cannot run on multiple threads (for example, if only one core is available or `NumThreads` is 0), MATLAB Coder executes the loop in a serial manner.

Limitations

- You must use a compiler that supports the Open Multiprocessing (OpenMP) application interface. See `https://www.mathworks.com/support/compilers/current_release/`. If you use a compiler that does not support OpenMP, MATLAB Coder treats the `parfor`-loops as `for`-loops. In the generated MEX function or C/C++ code, the loop iterations run on a single thread.
- The OpenMP application interface is not compatible with JIT MEX compilation. See "JIT Compilation Does Not Support OpenMP".
- Do not use the following constructs inside `parfor`-loops:
  - You cannot call extrinsic functions using `coder.extrinsic` in the body of a `parfor`-loop.
  - You cannot write to a global variable inside a `parfor`-loop.
  - MATLAB Coder does not support the use of `coder.ceval` in reductions. For example, you cannot generate code for the following `parfor`-loop:

    ```
    parfor i = 1:4
      y = coder.ceval('myCFcn',y,i);
    end
    ```

    Instead, write a local function that calls the C code using `coder.ceval` and call this function in the `parfor`-loop. For example:

    ```
    parfor i = 1:4
      y = callMyCFcn(y,i);
    end
    function y = callMyCFcn(y,i)
     y = coder.ceval('mCyFcn', y , i);
    end
    ```
  - You cannot use `varargin` or `varargout` in `parfor`-loops.
- The type of the loop index must be representable by an integer type on the target hardware. Use a type that does not require a multiword type in the generated code.
- `parfor` for standalone code generation requires the toolchain approach for building executables or libraries. Do not change settings that cause the code generator to use

the template makefile approach. See “Project or Configuration Is Using the Template Makefile”.

For a comprehensive list of restrictions, see “parfor Restrictions”.

Tips

- Use a `parfor`-loop when:
  - You need many loop iterations of a simple calculation. `parfor` divides the loop iterations into groups so that each thread can execute one group of iterations.
  - You have loop iterations that take a long time to execute.
- Do not use a `parfor`-loop when an iteration in your loop depends on the results of other iterations.

  Reductions are one exception to this rule. A reduction variable accumulates a value that depends on all the iterations together, but is independent of the iteration order.
- The input argument `NumThreads` sets the OpenMP `num_threads()` clause in the generated code. OpenMP also supports globally limiting the number of threads in C/C++ by setting the environment variable `OMP_NUM_THREADS` or by using `omp_set_num_threads()`. For more information, see the openMP specifications. https://www.openmp.org/specifications/

See Also

Functions
`codegen`

Topics
“Generate Code with Parallel for-Loops (parfor)”
“Algorithm Acceleration Using Parallel for-Loops (parfor)”
“Control Compilation of parfor-Loops”
“When to Use parfor-Loops”
“When Not to Use parfor-Loops”
“Classification of Variables in parfor-Loops”

Introduced in R2012b

addOption

Class: coder.make.BuildConfiguration
Package: coder.make

Add new option

Syntax

```
h.addOption(OptionName, buildItemHandle)
```

Description

`h.addOption(OptionName, buildItemHandle)` adds an option to `coder.make.BuildConfiguration.Options`.

Tips

Before using `addOption`, create a `coder.make.BuildItem` object to use as the second argument.

Input Arguments

`h` — BuildConfiguration handle
object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: h

`OptionName` — Name of option
new option name

Name of option, specified as a character vector. Choose a new option name.

Example: `'faster2'`

Data Types: `char`

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

Examples

Using the Option-related methods interactively

```
tc = coder.make.ToolchainInfo;
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.isOption('X Compiler')
```

```
ans  =

     0
```

```
bi = coder.make.BuildItem('WV','wrongvalue')
```

```
bi =

    Macro  : WV
    Value : wrongvalue
```

```
cfg.addOption('X Compiler',bi);
value = cfg.getOption('X Compiler')
```

```
value =

    Macro  : WV
    Value : wrongvalue
```

```
cfg.setOption('X Compiler','rightvalue');
value = cfg.getOption('X Compiler')
```

```
value =
```

```
Macro  : WV
Value : rightvalue
```

See Also

getOption | isOption | setOption | coder.make.BuildItem

getOption

Class: coder.make.BuildConfiguration
Package: coder.make

Get value of option

Syntax

```
OptionValue = h.getOption(OptionName)
```

Description

`OptionValue = h.getOption(OptionName)` returns the value and optional macro name of a build configuration option.

Input Arguments

h — BuildConfiguration handle

object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: h

OptionName — Name of option

new option name

Name of option, specified as a character vector. Choose a new option name.

Example: `'faster2'`

Data Types: `char`

Output Arguments

`OptionValue` — Value of option
coder.make.BuildItem object

Value of the option, returned as a `coder.make.BuildItem` object that contains a value and an optional macro name.

Examples

Using the Option-related methods interactively

```
tc = coder.make.ToolchainInfo;
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.isOption('X Compiler')

ans  =

     0

bi = coder.make.BuildItem('WV','wrongvalue')

bi =

    Macro  : WV
    Value : wrongvalue

cfg.addOption('X Compiler',bi);
value = cfg.getOption('X Compiler')

value =

    Macro  : WV
    Value : wrongvalue

cfg.setOption('X Compiler','rightvalue');
value = cfg.getOption('X Compiler')

value =

    Macro  : WV
    Value : rightvalue
```

See Also

addOption | isOption | setOption

info

Class: coder.make.BuildConfiguration
Package: coder.make

Get information about build configuration

Syntax

```
h.info
OutputInfo = h.info
```

Description

`h.info` displays information about the `coder.make.BuildConfiguration` object in the MATLAB Command Window.

`OutputInfo = h.info` returns information about the `coder.make.BuildConfiguration` object

Input Arguments

h — BuildConfiguration handle
object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: h

Output Arguments

`OutputInfo` — Build configuration information
character vector

Build configuration information, returned as a character vector.

Examples

The `intel_tc.m` file from “Adding a Custom Toolchain”, uses the following lines to display information about the `BuildConfiguration` property:

```
tc = intel_tc
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.info
```

```
############################################
# Build Configuration : Faster Builds
# Description         : Default settings for faster compile/link of code
############################################

ARFLAGS              = /nologo
CFLAGS               = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /Od
CPPFLAGS             = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /Od
DOWNLOAD_FLAGS        =
EXECUTE_FLAGS         =
LDFLAGS              = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN)
MEX_CFLAGS            =
MEX_LDFLAGS           =
MAKE_FLAGS           = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS    = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE)
```

isOption

Class: coder.make.BuildConfiguration
Package: coder.make

Determine if option exists

Syntax

```
OutputValue = isOption(OptionName)
```

Description

`OutputValue = isOption(OptionName)` returns `'1'` (true) if the specified option exists. Otherwise, it returns `'0'` (false).

Input Arguments

h — BuildConfiguration handle

object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: h

OptionName — Name of option

new option name

Name of option, specified as a character vector. Choose a new option name.

Example: `'faster2'`

Data Types: `char`

Output Arguments

`OutputValue` — Option exists
`0 | 1`

Option exists, returned as a logical value. If the option exists, the value is `'1'` (true). Otherwise, the output is `'0'` (false).

Examples

Using the Option-related methods interactively

```
tc = coder.make.ToolchainInfo;
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.isOption('X Compiler')

ans  =

     0

bi = coder.make.BuildItem('WV','wrongvalue')

bi =

    Macro  : WV
    Value : wrongvalue

cfg.addOption('X Compiler',bi);
value = cfg.getOption('X Compiler')

value =

    Macro  : WV
    Value : wrongvalue

cfg.setOption('X Compiler','rightvalue');
value = cfg.getOption('X Compiler')

value =

    Macro  : WV
    Value : rightvalue
```

See Also

`addOption` | `getOption` | `setOption` | `coder.make.BuildItem`

keys

Class: coder.make.BuildConfiguration
Package: coder.make

Get all option names

Syntax

```
Out = h.keys
```

Description

`Out = h.keys` returns a list of all option names or keys in the build configuration.

Input Arguments

h — BuildConfiguration handle

object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: h

Output Arguments

Output — List of all option names or keys in build configuration

cell array of character vectors

List of all option names or keys in build configuration, returned as a cell array of character vectors.

Examples

The `intel_tc.m` file from “Adding a Custom Toolchain”, uses the following lines to display keys from the `BuildConfiguration` property:

```
tc = intel_tc
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.keys

ans =

  Columns 1 through 5

    'Archiver'    'C Compiler'    'C++ Compiler'    'Download'    'Execute'

  Columns 6 through 10

    'Linker'    'MEX Compiler'    'MEX Linker'    'Make Tool'    [1x21 char]
```

setOption

Class: coder.make.BuildConfiguration
Package: coder.make

Set value of option

Syntax

```
h.setOption(OptionName, OptionValue)
```

Description

`h.setOption(OptionName, OptionValue)` updates the values within a coder.make.BuildConfiguration object.

Input Arguments

h — BuildConfiguration handle

object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: `h`

OptionName — Name of option

new option name

Name of option, specified as a character vector. Choose a new option name.

Example: `'faster2'`

Data Types: `char`

OptionValue — Value of option

character vector or object handle

Value of option, specified as a character vector, or as the handle of a `coder.make.BuildItem` object that contains an option value.

Example: `linkerOpts`

Examples

The setOption method in intel_tc

The `intel_tc.m` file from “Adding a Custom Toolchain”, gets a default `BuildConfiguration` object and then uses `setOption` to update the values in that object:

```
% ------------------------------------------
% BUILD CONFIGURATIONS
% ------------------------------------------

optimsOffOpts = {'/c /Od'};
optimsOnOpts = {'/c /O2'};
cCompilerOpts    = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL)';
cppCompilerOpts  = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL)';
linkerOpts       = {'$(ldebug) $(conflags) $(LIBS_TOOLCHAIN)'};
sharedLinkerOpts = horzcat(linkerOpts,'-dll -def:$(DEF_FILE)');
archiverOpts     = {'/nologo'};

% Get the debug flag per build tool
debugFlag.CCompiler   = '$(CDEBUG)';
debugFlag.CppCompiler = '$(CPPDEBUG)';
debugFlag.Linker      = '$(LDDEBUG)';
debugFlag.Archiver    = '$(ARDEBUG)';

cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOffOpts));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOffOpts));
cfg.setOption('Linker',linkerOpts);
cfg.setOption('Shared Library Linker',sharedLinkerOpts);
cfg.setOption('Archiver',archiverOpts);

cfg = tc.getBuildConfiguration('Faster Runs');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOnOpts));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOnOpts));
cfg.setOption('Linker',linkerOpts);
cfg.setOption('Shared Library Linker',sharedLinkerOpts);
cfg.setOption('Archiver',archiverOpts);

cfg = tc.getBuildConfiguration('Debug');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOffOpts,debugFlag.CCompiler));
cfg.setOption ...
('C++ Compiler',horzcat(cppCompilerOpts,optimsOffOpts,debugFlag.CppCompiler));
cfg.setOption('Linker',horzcat(linkerOpts,debugFlag.Linker));
cfg.setOption('Shared Library Linker',horzcat(sharedLinkerOpts,debugFlag.Linker));
cfg.setOption('Archiver',horzcat(archiverOpts,debugFlag.Archiver));

tc.setBuildConfigurationOption('all','Download','');
```

```
tc.setBuildConfigurationOption('all','Execute','');
tc.setBuildConfigurationOption('all','Make Tool','-f $(MAKEFILE)');
```

Using the Option-related methods interactively

```
tc = coder.make.ToolchainInfo;
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.isOption('X Compiler')

ans  =

     0

bi = coder.make.BuildItem('WV','wrongvalue')

bi =

    Macro  : WV
    Value : wrongvalue

cfg.addOption('X Compiler',bi);
value = cfg.getOption('X Compiler')

value =

    Macro  : WV
    Value : wrongvalue

cfg.setOption('X Compiler','rightvalue');
value = cfg.getOption('X Compiler')

value =

    Macro  : WV
    Value : rightvalue
```

See Also

`addOption` | `getOption` | `isOption` | `coder.make.BuildItem`

Topics

“Adding a Custom Toolchain”

values

Class: coder.make.BuildConfiguration
Package: coder.make

Get all option values

Syntax

```
Out = h.values
```

Description

`Out = h.values` returns a list of all option values in the build configuration.

Input Arguments

h — BuildConfiguration handle

object handle

BuildConfiguration handle, specified as a `coder.make.BuildConfiguration` object.

Example: h

Output Arguments

Out — List of all option values in build configuration

character vector or object handle

List of all option values in the build configuration, returned as a cell array of character vectors.

Examples

Starting from the “Adding a Custom Toolchain” example, enter the following lines:

```
tc = intel_tc
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.values
```

```
ans =

  Columns 1 through 2

    [1x1 coder.make.BuildItem]    [1x1 coder.make.BuildItem]

  Columns 3 through 4

    [1x1 coder.make.BuildItem]    [1x1 coder.make.BuildItem]

  Columns 5 through 6

    [1x1 coder.make.BuildItem]    [1x1 coder.make.BuildItem]

  Columns 7 through 8

    [1x1 coder.make.BuildItem]    [1x1 coder.make.BuildItem]

  Columns 9 through 10

    [1x1 coder.make.BuildItem]    [1x1 coder.make.BuildItem]
```

getMacro

Class: coder.make.BuildItem
Package: coder.make

Get macro name of build item

Syntax

```
h.getMacro
```

Description

`h.getMacro` returns the macro name of an existing build item.

Input Arguments

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

Examples

```
bi = coder.make.BuildItem('bldtmvalue')
```

```
bi =

    Macro  : (empty)
    Value : bldtmvalue
```

```
bi.setMacro('BIMV2');
bi.getMacro
```

```
ans =

    BIMV2
```

See Also

`getMacro` | `getValue` | `setMacro` | `setValue`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getValue

Class: coder.make.BuildItem
Package: coder.make

Get value of build item

Syntax

```
h.getValue
```

Description

`h.getValue` returns the value of an existing build item.

Input Arguments

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

Examples

```
bi = coder.make.BuildItem('wrongvalue')
```

```
bi =

    Macro  : (empty)
    Value : wrongvalue
```

```
bi.setValue('rightvalue')
bi.getValue
```

```
ans  =

rightvalue
```

See Also

`getMacro` | `getValue` | `setMacro` | `setValue`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setMacro

Class: coder.make.BuildItem
Package: coder.make

Set macro name of build item

Syntax

```
h.setMacro(blditm_macroname)
```

Description

`h.setMacro(blditm_macroname)` sets the macro name of an existing build item.

Input Arguments

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

`blditm_macroname` — Macro name of build item
character vector

Macro name of build item, specified as a character vector.

Data Types: `char`

Examples

```
bi = coder.make.BuildItem('bldtmvalue')
```

```
bi =

    Macro  : (empty)
    Value : bldtmvalue

bi.setMacro('BIMV2');
bi.getMacro

ans =

    BIMV2
```

See Also

`getMacro` | `getValue` | `setMacro` | `setValue`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setValue

Class: coder.make.BuildItem
Package: coder.make

Set value of build item

Syntax

```
h.setValue(blditm_value)
```

Description

`h.setValue(blditm_value)` sets the value of an existing build item macro.

Input Arguments

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

`blditm_value` — Value of build item
character vector

Value of build item

Data Types: `char`

Examples

```
bi = coder.make.BuildItem('wrongvalue')
```

```
bi =

    Macro  : (empty)
    Value : wrongvalue

bi.setValue('rightvalue')
bi.getValue

ans  =

rightvalue
```

See Also

`getMacro` | `getValue` | `setMacro` | `setValue`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addDirective

Class: coder.make.BuildTool
Package: coder.make

Add directive to `Directives`

Syntax

```
h.addDirective(name,value)
```

Description

`h.addDirective(name,value)` creates a named directive, assigns a value to it, and adds it to `coder.make.BuildTool.Directives`.

Input Arguments

`h` — Object handle
variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`name` — Name of directive
character vector

Name of directive, specified as a character vector.

Data Types: `char`

`value` — Value of directive
character vector

Value of directive, specified as a character vector.

Data Types: `char`

Examples

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.addDirective('IncludeSearchPath','-O');
tool.setDirective('IncludeSearchPath','-I');
tool.getDirective('IncludeSearchPath')

ans  =

-I
```

See Also

“Properties” on page 3-83 | `getDirective` | `setDirective`

Topics

“Example” on page 3-87

addFileExtension

Class: coder.make.BuildTool
Package: coder.make

Add new file extension entry to `FileExtensions`

Syntax

```
h.addFileExtension(name,buildItemHandle)
```

Description

`h.addFileExtension(name,buildItemHandle)` creates a named extension, assigns a `coder.make.BuildItem` object to it, and adds it to `coder.make.BuildTool.FileExtensions`.

Input Arguments

`h` — Object handle
variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`name` — Name of file type.
character vector

Name of file type, specified as a character vector.

Data Types: `char`

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

Examples

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
bldtm = coder.make.BuildItem('CD','.cd')
```

```
bldtm =

    Macro  : CD
    Value : .cd
```

```
tool.addFileExtension('SourceX',bldtm)
value = tool.getFileExtension('SourceX')
```

```
value  =

.cd
```

```
tool.setFileExtension('SourceX','.ef')
value = tool.getFileExtension('SourceX')
```

```
value  =

.ef
```

See Also

“Properties” on page 3-83 | `getFileExtension` | `setFileExtension`

Topics

“Adding a Custom Toolchain”

getCommand

Class: coder.make.BuildTool
Package: coder.make

Get build tool command

Syntax

```
c_out = h.getCommand
c_out = h.getCommand('value')
c_out = h.getCommand('macro')
```

Description

`c_out = h.getCommand` returns the value of the `coder.make.BuildTool.Command` property.

`c_out = h.getCommand('value')` also returns the value of `coder.make.BuildTool.Command`.

`c_out = h.getCommand('macro')` returns the macro name of `coder.make.BuildTool.Command`.

Input Arguments

h — Object handle
variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`'value'` — Get command value
character vector

Gets the command value.

`'macro'` — Get command macro
character vector

Gets the command macro.

Output Arguments

`c_out` — Command value or macro
variable

The command value or macro of the build tool, returned as a scalar.

Data Types: `char`

Examples

```
tc = coder.make.ToolchainInfo;
btl = tc.getBuildTool('C Compiler');
btl.getCommand
```

```
ans  =
```

```
icl
```

```
btl.getCommand('value')
```

```
ans  =
```

```
icl
```

```
c_out = btl.getCommand('macro')
```

```
c_out  =
```

```
CC
```

See Also

`setCommand`

getDirective

Class: coder.make.BuildTool
Package: coder.make

Get value of named directive from `Directives`

Syntax

```
value = h.getDirective(name)
```

Description

`value = h.getDirective(name)` gets the value of the named directive from `Directives`

Input Arguments

`h` — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`name` — Name of directive

character vector

Name of directive, specified as a character vector.

Data Types: `char`

Output Arguments

`value` — Value of directive

character vector

Value of directive, specified as a character vector.

Data Types: `char`

Examples

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.addDirective('IncludeSearchPath','-O');
tool.setDirective('IncludeSearchPath','-I');
tool.getDirective('IncludeSearchPath')
```

```
ans  =

-I
```

See Also

“Properties” on page 3-83 | `addDirective` | `setDirective`

Topics

“Adding a Custom Toolchain”

getFileExtension

Class: coder.make.BuildTool
Package: coder.make

Get file extension for named file type in `FileExtensions`

Syntax

```
value = h.getFileExtension(name)
```

Description

`value = h.getFileExtension(name)` gets the file extension of the named file type from `coder.make.BuildTool.FileExtensions`.

Input Arguments

`h` — Object handle
variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`name` — Name of file type.
character vector

Name of file type, specified as a character vector.

Data Types: `char`

Output Arguments

`value` — Value of file extension
character vector

Value of file extension, specified as a character vector.

Data Types: `char`

Examples

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
bldtm = coder.make.BuildItem('CD','.cd')
```

```
bldtm =

    Macro  : CD
    Value : .cd
```

```
tool.addFileExtension('SourceX',bldtm)
value = tool.getFileExtension('SourceX')
```

```
value  =

.cd
```

```
tool.setFileExtension('SourceX','.ef')
value = tool.getFileExtension('SourceX')
```

```
value  =

.ef
```

See Also

“Properties” on page 3-83 | `addFileExtension` | `setFileExtension`

Topics

“Adding a Custom Toolchain”

getName

Class: coder.make.BuildTool
Package: coder.make

Get build tool name

Syntax

```
toolname = h.getName
```

Description

`toolname = h.getName` returns the name of the `coder.make.BuildTool` object.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

Output Arguments

toolname — Name of BuildTool object

The name of the `coder.make.BuildTool` object

Examples

Using the getName and setName methods interactively

Starting from the “Adding a Custom Toolchain” example, enter the following lines:

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.getName
```

```
ans  =

C Compiler
```

```
tool.setName('X Compiler')
tool.getName
```

```
ans  =

X Compiler
```

See Also

setName

getPath

Class: coder.make.BuildTool
Package: coder.make

Get path and macro of build tool in `Path`

Syntax

```
btpath = h.getPath
btmacro = h.getPath('macro')
```

Description

`btpath = h.getPath` returns the path of the build tool from **coder.make.BuildTool.Paths**.

`btmacro = h.getPath('macro')` returns the macro for the path of the build tool from **coder.make.BuildTool.Paths**

Tips

If the system command environment specifies a path variable for the build tool, the value of the path does not need to be specified by the `BuildTool` object.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

Output Arguments

`btpath` — Path of build tool object
character vector

The path of `BuildTool` object, returned as a scalar.

Data Types: `char`

`btmacro` — Macro for path of build tool object
character vector

Macro for path of `BuildTool` object, returned as a scalar.

Data Types: `char`

Examples

Enter the following lines:

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.getPath

ans  =

     ''

tool.getPath('macro')

ans  =

CC_PATH

tool.setPath('/gcc')
tool.Path

ans =

    Macro  : CC_PATH
    Value : /gcc
```

See Also

“Properties” on page 3-83 | `setPath`

Topics

“Adding a Custom Toolchain”

info

Class: coder.make.BuildTool
Package: coder.make

Display build tool properties and values

Syntax

`h.info`

Description

`h.info` returns information about the `coder.make.BuildTool` object.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

Examples

Starting from the “Adding a Custom Toolchain” example, enter the following lines:

```
tc = intel_tc;
tool = tc.getBuildTool('C Compiler');
tool.info
```

```
###########################################
# Build Tool: Intel C Compiler
###########################################

Language              : 'C'
OptionsRegistry       : {'C Compiler','CFLAGS'}
InputFileExtensions   : {'Source'}
```

```
OutputFileExtensions  : {'Object'}
DerivedFileExtensions : {'|>OBJ_EXT<|'}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
CC = icl
CC_PATH  =

# ------------
# Directives
# ------------
Debug              = -Zi
Include            =
IncludeSearchPath  = -I
OutputFlag         = -Fo
PreprocessorDefine = -D

# -----------------
# File Extensions
# -----------------
Header = .h
Object = .obj
Source = .c
```

See Also

Topics

“Adding a Custom Toolchain”

setCommand

Class: coder.make.BuildTool
Package: coder.make

Set build tool command

Syntax

```
h.setCommand(commandvalueinput)
```

Description

`h.setCommand(commandvalueinput)` sets the value of the `coder.make.BuildTool.Command` property.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

commandvalueinput — Value of coder.make.BuildTool.Command property

character vector

Value of the `coder.make.BuildTool.Command` property. Enter a character vector, or the handle of a `coder.make.BuildItem` object that contains an option value.

Examples

Get a default build tool and set its properties

The `intel_tc.m` file from "Adding a Custom Toolchain", uses the following lines to set the command of a default build tool, `C Compiler`, from a `ToolchainInfo` object called `tc`, and then sets its properties.

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

See Also

`getCommand`

Topics

"Toolchain Definition File with Commentary"
"Adding a Custom Toolchain"

setCommandPattern

Class: coder.make.BuildTool
Package: coder.make

Set pattern of commands for build tools

Syntax

```
h.setCommandPattern(commandpattern);
```

Description

`h.setCommandPattern(commandpattern);` sets the command pattern of a specific `coder.make.BuildTool` object in `coder.make.ToolchainInfo.BuildTools`.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

commandpattern — Pattern of commands and options

character vector

Pattern of commands and options that a `BuildTool` can use to run a build tool, specified as a character vector.

Use `|>` and `<|` as the left and right delimiters of a command element. Use a space character between the `<|` and `|>` delimiters to require a space between two command elements. For example:

- `|>TOOL<| |>TOOL_OPTIONS<|` requires a space between the two command elements.

- `|>OUTPUT_FLAG<||>OUTPUT<|` requires no space between the two command elements.

Data Types: `char`

Examples

The `intel_tc.m` file from "Adding a Custom Toolchain", uses the following lines to get and update one of the `BuildTool` objects, including the command pattern:

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

See Also

`addBuildTool` | `setCommandPattern`

Topics

"Adding a Custom Toolchain"

setCompilerOptionMap

Class: coder.make.BuildTool
Package: coder.make

Set C/C++ language standard and compiler options for selected build tool (compiler)

Syntax

```
h.setCompilerOptionMap(std,opts);
```

Description

`h.setCompilerOptionMap(std,opts);` sets the C/C++ language standard and corresponding compiler options of a specific `coder.make.BuildTool` object in `coder.make.ToolchainInfo.BuildTools`.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

std — C/C++ language standard

character vector

The C/C++ language standard for the compiler in the `BuildTool` definition, specified as a character vector.

Use one of the following for the `std` value:

- `'C89/C90 (ANSI)'`

- `'C99 (ISO)'`
- `'C++03 (ISO)'`

Data Types: `char`

`opts` — Compiler options
character vector

The compiler options correspond to specific `BuildTool` name and C/C++ language standard selections, specified as a character vector. The table provides example value combinations for an Intel toolchain.

Build Tool Name (Compiler)	std Value	opts Value
`'Intel C Compiler'`	`'C99 (ISO)'`	`'/Qstd=c99'` for Windows `'-std=c99'` for UNIX®
`'Intel C++ Compiler'`	`'C++03 (ISO)'`	`'/Qstd=c++0x'` for Windows `'-std=c++0x'` for UNIX

For more Intel-related options, visit `https://software.intel.com/en-us/articles/iso-iec-standards-language-conformance-for-intel-c-compiler`:

Data Types: `char`

Examples

The following version of the `intel_tc.m` file differs from the “Adding a Custom Toolchain” example. This example shows how to define the `C_STANDARD_OPTS` and `CPP_STANDARD_OPTS` macros, set values for the macros with the `setCompilerOptionMap` method, and apply the macros in build configurations.

Define C/C++ STANDARD_OPTS Macros

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding a build tool to ToolchainInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function tc = intel_tc
%INTEL_TC Creates an Intel v14 ToolchainInfo object.
% This file can be used as a template to define other toolchains on Windows.
```

```
% Copyright 2012-2016 The MathWorks, Inc.

tc = coder.make.ToolchainInfo('BuildArtifact', 'nmake makefile');
tc.Name             = 'Intel v14 | nmake makefile (64-bit Windows)';
tc.Platform         = 'win64';
tc.SupportedVersion = '14';

tc.addAttribute('TransformPathsWithSpaces');
tc.addAttribute('RequiresCommandFile');
tc.addAttribute('RequiresBatchFile');

% ------------------------------
% Setup
% ------------------------------
% Below we are using %ICPP_COMPILER14% as root folder where Intel Compiler is installed.
% You can either set an environment variable or give full path to the
% compilervars.bat file
tc.ShellSetup{1} = 'call %ICPP_COMPILER14%\bin\compilervars.bat intel64';


% ------------------------------
% Macros
% ------------------------------
tc.addMacro('MW_EXTERNLIB_DIR',     ['$(MATLAB_ROOT)\extern\lib\' tc.Platform '\microsoft']);
tc.addMacro('MW_LIB_DIR',           ['$(MATLAB_ROOT)\lib\' tc.Platform]);
tc.addMacro('CFLAGS_ADDITIONAL',    '-D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('CPPFLAGS_ADDITIONAL',  '-EHs -D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('LIBS_TOOLCHAIN',       '$(conlibs)');
tc.addMacro('CVARSFLAG',            '');

tc.addIntrinsicMacros({'ldebug', 'conflags', 'cflags'});
tc.addIntrinsicMacros({'C_STANDARD_OPTS', 'CPP_STANDARD_OPTS'});

% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName(           'Intel C Compiler');
tool.setCommand(        'icl');
tool.setPath(           '');

tool.setDirective(      'IncludeSearchPath',    '-I');
tool.setDirective(      'PreprocessorDefine',   '-D');
tool.setDirective(      'OutputFlag',           '-Fo');
tool.setDirective(      'Debug',                '-Zi');

tool.setFileExtension(  'Source',               '.c');
tool.setFileExtension(  'Header',               '.h');
tool.setFileExtension(  'Object',               '.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

```
tool.setCompilerOptionMap('C99 (ISO)', '/Qstd=c99');

% ------------------------------
% C++ Compiler
% ------------------------------

tool = tc.getBuildTool('C++ Compiler');

tool.setName(           'Intel C++ Compiler');
tool.setCommand(        'icl');
tool.setPath(           '');

tool.setDirective(      'IncludeSearchPath',      '-I');
tool.setDirective(      'PreprocessorDefine',     '-D');
tool.setDirective(      'OutputFlag',           '-Fo');
tool.setDirective(      'Debug',                '-Zi');

tool.setFileExtension(  'Source',               '.cpp');
tool.setFileExtension(  'Header',               '.hpp');
tool.setFileExtension(  'Object',               '.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

tool.setCompilerOptionMap('C99 (ISO)', '/Qstd=c++0x');

% ------------------------------
% Linker
% ------------------------------

tool = tc.getBuildTool('Linker');

tool.setName(           'Intel C/C++ Linker');
tool.setCommand(        'xilink');
tool.setPath(           '');

tool.setDirective(      'Library',              '-L');
tool.setDirective(      'LibrarySearchPath',    '-I');
tool.setDirective(      'OutputFlag',           '-out:');
tool.setDirective(      'Debug',                '');

tool.setFileExtension(  'Executable',           '.exe');
tool.setFileExtension(  'Shared Library',       '.dll');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

% ------------------------------
% C++ Linker
% ------------------------------

tool = tc.getBuildTool('C++ Linker');

tool.setName(           'Intel C/C++ Linker');
tool.setCommand(        'xilink');
tool.setPath(           '');
```

```
tool.setDirective(      'Library',              '-L');
tool.setDirective(      'LibrarySearchPath',    '-I');
tool.setDirective(      'OutputFlag',           '-out:');
tool.setDirective(      'Debug',                '');

tool.setFileExtension(  'Executable',           '.exe');
tool.setFileExtension(  'Shared Library',       '.dll');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

% ------------------------------
% Archiver
% ------------------------------

tool = tc.getBuildTool('Archiver');

tool.setName(           'Intel C/C++ Archiver');
tool.setCommand(        'xilib');
tool.setPath(           '');
tool.setDirective(      'OutputFlag',           '-out:');
tool.setFileExtension(  'Static Library',       '.lib');
tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

% ------------------------------
% Builder
% ------------------------------

tc.setBuilderApplication(tc.Platform);

% --------------------------------------------
% BUILD CONFIGURATIONS
% --------------------------------------------

optimsOffOpts    = {'/c /Od'};
optimsOnOpts     = {'/c /O2'};
cCompilerOpts    = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) $(C_STANDARD_OPTS) ';
cppCompilerOpts  = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) $(CPP_STANDARD_OPTS)';
linkerOpts       = {'$(ldebug) $(conflags) $(LIBS_TOOLCHAIN)'};
sharedLinkerOpts = horzcat(linkerOpts, '-dll -def:$(DEF_FILE)');
archiverOpts     = {'/nologo'};

% Get the debug flag per build tool
debugFlag.CCompiler   = '$(CDEBUG)';
debugFlag.CppCompiler = '$(CPPDEBUG)';
debugFlag.Linker      = '$(LDDEBUG)';
debugFlag.CppLinker   = '$(CPPLDDEBUG)';
debugFlag.Archiver    = '$(ARDEBUG)';

% Set the toolchain flags for 'Faster Builds' build configuration

cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption( 'C Compiler',               horzcat(cCompilerOpts,   optimsOffOpts));
cfg.setOption( 'C++ Compiler',             horzcat(cppCompilerOpts, optimsOffOpts));
cfg.setOption( 'Linker',                   linkerOpts);
cfg.setOption( 'C++ Linker',               linkerOpts);
```

```
cfg.setOption( 'Shared Library Linker',     sharedLinkerOpts);
cfg.setOption( 'Archiver',                  archiverOpts);

% Set the toolchain flags for 'Faster Runs' build configuration

cfg = tc.getBuildConfiguration('Faster Runs');
cfg.setOption( 'C Compiler',                horzcat(cCompilerOpts,   optimsOnOpts));
cfg.setOption( 'C++ Compiler',              horzcat(cppCompilerOpts, optimsOnOpts));
cfg.setOption( 'Linker',                    linkerOpts);
cfg.setOption( 'C++ Linker',                linkerOpts);
cfg.setOption( 'Shared Library Linker',     sharedLinkerOpts);
cfg.setOption( 'Archiver',                  archiverOpts);

% Set the toolchain flags for 'Debug' build configuration

cfg = tc.getBuildConfiguration('Debug');
cfg.setOption( 'C Compiler',                  horzcat(cCompilerOpts,   optimsOffOpts, debugFlag.CCompiler));
cfg.setOption( 'C++ Compiler',              horzcat(cppCompilerOpts, optimsOffOpts, debugFlag.CppCompiler));
cfg.setOption( 'Linker',                    horzcat(linkerOpts,       debugFlag.Linker));
cfg.setOption( 'C++ Linker',                   horzcat(linkerOpts,       debugFlag.CppLinker));
cfg.setOption( 'Shared Library Linker',      horzcat(sharedLinkerOpts, debugFlag.Linker));
cfg.setOption( 'Archiver',                  horzcat(archiverOpts,     debugFlag.Archiver));

tc.setBuildConfigurationOption('all', 'Download',      '');
tc.setBuildConfigurationOption('all', 'Execute',       '');
tc.setBuildConfigurationOption('all', 'Make Tool',     '-f $(MAKEFILE)');
```

See Also

`coder.make.BuildTool`

Topics

“Adding a Custom Toolchain”

setDirective

Class: coder.make.BuildTool
Package: coder.make

Set value of directive in `Directives`

Syntax

```
h.setDirective(name,value)
```

Description

`h.setDirective(name,value)` assigns a value to the named directive in `coder.make.Directives`.

Input Arguments

`h` — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`name` — Name of directive

character vector

Name of directive, specified as a character vector.

Data Types: `char`

`value` — Value of directive

character vector

Value of directive, specified as a character vector.

Data Types: `char`

Examples

Get a default build tool and set its properties

The following example code shows `setDirective` in a portion of the `intel_tc.m` file from the "Adding a Custom Toolchain" tutorial.

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

Use the setDirective method interactively

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.addDirective('IncludeSearchPath','-O');
tool.setDirective('IncludeSearchPath','-I');
tool.getDirective('IncludeSearchPath')
```

```
ans  =

-I
```

See Also

"Properties" on page 3-83 | `addDirective` | `getDirective`

Topics

“Adding a Custom Toolchain”

setFileExtension

Class: coder.make.BuildTool
Package: coder.make

Set file extension for named file type in `FileExtensions`

Syntax

```
h.setFileExtension(name,value)
```

Description

`h.setFileExtension(name,value)` sets the extension value of the named file type in `coder.make.BuildTool.FileExtensions`.

Input Arguments

`h` — Object handle
variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`name` — Name of file type.
character vector

Name of file type, specified as a character vector.

Data Types: `char`

`value` — Value of file extension
character vector

Value of file extension, specified as a character vector.

Data Types: `char`

Examples

Get a default build tool and set its properties

The following example code shows `setFileExtension` in a portion of the `intel_tc.m` file from the "Adding a Custom Toolchain" tutorial.

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

Use the setFileExtension interactively

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
bldtm = coder.make.BuildItem('CD','.cd')

bldtm =

    Macro  : CD
    Value : .cd

tool.addFileExtension('SourceX',bldtm)
value = tool.getFileExtension('SourceX')

value  =

.cd
```

```
tool.setFileExtension('SourceX','.ef')
value = tool.getFileExtension('SourceX')
```

```
value  =

.ef
```

See Also

“Properties” on page 3-83 | `addFileExtension` | `getFileExtension`

Topics

“Adding a Custom Toolchain”

setName

Class: coder.make.BuildTool
Package: coder.make

Set build tool name

Syntax

```
h.setName(name)
```

Description

`h.setName(name)` sets the name of the `coder.make.BuildTool.Name` property.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

name — Name of build tool

character vector

The name of the build tool, specified as a character vector.

Example: `'Intel C Compiler'`

Data Types: `char`

Examples

Get a default build tool and set its properties

The following example code shows `setName` in a portion of the `intel_tc.m` file from the "Adding a Custom Toolchain" tutorial:

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

Using the getName and setName methods interactively

Starting from the "Adding a Custom Toolchain" example, enter the following lines:

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.getName

ans  =

C Compiler

tool.setName('X Compiler')
tool.getName

ans  =

X Compiler
```

See Also

Topics

“Toolchain Definition File with Commentary”
“Adding a Custom Toolchain”

setPath

Class: coder.make.BuildTool
Package: coder.make

Set path and macro of build tool in `Path`

Syntax

```
h.setPath(btpath,btmacro)
```

Description

`h.setPath(btpath,btmacro)` sets the path and macro of the build tool in **coder.make.BuildTool.Paths**.

Input Arguments

`h` — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

`btpath` — Path of build tool object

character vector

The path of `BuildTool` object, returned as a scalar.

Data Types: `char`

`btmacro` — Macro for path of build tool object

character vector

Macro for path of `BuildTool` object, returned as a scalar.

Data Types: `char`

Examples

Get a default build tool and set its properties

The following example code shows `setPath` in a portion of the `intel_tc.m` file from the "Adding a Custom Toolchain" tutorial.

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

Use the getPath and setPath methods interactively

This example shows example inputs and outputs for the methods in a MATLAB Command Window:

Enter the following lines:

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
tool.getPath

ans  =

     ''

tool.getPath('macro')
```

```
ans  =

CC_PATH
```

```
tool.setPath('/gcc')
tool.Path
```

```
ans =

    Macro  : CC_PATH
    Value : /gcc
```

See Also

“Properties” on page 3-83 | `getPath`

Topics

“Adding a Custom Toolchain”

validate

Class: coder.make.BuildTool
Package: coder.make

Validate build tool properties

Syntax

```
validtool = h.validate
```

Description

`validtool = h.validate` validates the `coder.make.BuildTool` object, and generates errors if any properties are incorrectly defined.

Input Arguments

h — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

Output Arguments

validtool — Validity of coder.make.BuildTool object

The validity of the `coder.make.BuildTool` object. If the method detects a problem it returns `'0'` or an error message.

Examples

The `coder.make.BuildTool.validate` method returns warning and error messages if you try to validate a build tool before you have installed the build tool software (compiler, linker, archiver).

Starting from the "Adding a Custom Toolchain" example, enter the following lines:

```
tc = intel_tc;
tool = tc.getBuildTool('C Compiler');
tool.validate
```

If your host computer does not have the Intel toolchain installed, `validate` displays the following messages:

```
Warning: Validation of build tool 'Intel C Compiler' may require the toolchain
to be set up first. The setup information is registered in the toolchain
this build tool belong to. Pass the parent ToolchainInfo object to VALIDATE
in order for any toolchain setup to be done before validation.
> In C:\Program Files\MATLAB\R2013a\toolbox\coder\foundation\build\+coder\+make\
BuildTool.p>BuildTool.checkForPresence at 634
  In C:\Program Files\MATLAB\R2013a\toolbox\coder\foundation\build\+coder\+make\
BuildTool.p>BuildTool.validate at 430
Error using message
In 'CoderFoundation:toolchain:ValidateBuildToolError',data type supplied is
incorrect for parameter {1}.

Error in C:\Program
Files\MATLAB\R2013a\toolbox\coder\foundation\build\+coder\+make\
BuildTool.p>BuildTool.checkForPresence
(line 664)


Error in C:\Program
Files\MATLAB\R2013a\toolbox\coder\foundation\build\+coder\+make\
BuildTool.p>BuildTool.validate
(line 430)


Trial>>
```

For more information, see "Troubleshooting Custom Toolchain Validation".

See Also

`validate`

Topics

"Adding a Custom Toolchain"

“Troubleshooting Custom Toolchain Validation”

addAttribute

Class: coder.make.ToolchainInfo
Package: coder.make

Add custom attribute to `Attributes`

Syntax

```
h.addAttribute(Name,Value)
```

Description

`h.addAttribute(Name,Value)` adds a custom attribute with the specified name and value to `coder.make.ToolchainInfo.Attributes`. If you do not specify a value, the function initializes the attribute to `true` (default).

All attributes are optional. The toolchain uses the attributes during the build process.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Name-Value Pair Argument

Specify a `Name,Value` pair argument. `Name` is the argument name and `Value` is the corresponding value.

`TransformPathsWithSpaces` — Path spaces
true (default) | false

Look for spaces in paths to source files, include files, include paths, additional source paths, object paths, and prebuild object paths, library paths, and within MACROS used in any of the stated paths. If any path contains spaces, an alternate version of the path is returned. For long path names or paths with spaces, the method returns the '~' version on Windows when 8.3 name creation is enabled. On Linux platforms, paths with spaces are returned with the spaces escaped.

`RequiresBatchFile` — Batch file
true (default) | false

Create a batch file that runs the generated makefile.

`SupportsUNCPaths` — UNC paths
true (default) | false

Looks in the same locations for UNC paths (Windows only, ignored on Linux/Mac platforms). If there is a drive mapped to the UNC the path is pointing to, then paths that are UNC paths will have a mapped drive letter put in place.

`SupportsDoubleQuotes` — Double quotes
true (default) | false

Wrap each path in double quotes if they contain spaces.

`RequiresCommandFile` — Long compiler or archiver/linker lines
true (default) | false

For handling long compiler or archiver/linker lines (typically in Windows). If specified:

- In the compiler command line, the makefile replaces a long list of include paths with a call to the command file
- In the linker/archiver command line, the makefile replaces a long object file list with a call to a command file.

`NoCompilerCommandFile` — No compiler command file
true (default) | false

Build process does not create a response file for the header file paths in the model reference hierarchy, which modifies the behaviour specified by `RequiresCommandFile`. Set `NoCompilerCommandFile` to `true` only if you need to avoid long compiler command lines and your compiler does not support compiler command files. Use with the `CopyReferencedModelHeaders` attribute.

`CopyReferencedModelHeaders` — Copy model reference header files
true (default) | false

Copy model reference header files to the `referenced_model_includes` subfolder in the top model build folder. Set `CopyReferencedModelHeaders` to `true` only if you need to avoid long compiler command lines and your compiler does not support compiler command files. Use with the `NoCompilerCommandFile` attribute.

`LinkerLibraryPrefix` — Linker library prefix
character vector | string

Append prefix to the linker libraries specified on the command line. Applies only if toolchain has a `gmake` build artifact.

Example: `tc.addAttribute('LinkerLibraryPrefix', '--library=')`

Examples

Add an attribute and initialize its value, overriding the default value

```
h.Attribute
```

```
ans =


# -------------------
# "Attribute" List
# -------------------
(empty)
```

```
h.addAttribute('TransformPathsWithSpaces',false)
h.getAttribute('TransformPathsWithSpaces')
```

```
ans  =

     0
```

Add attribute without overriding its default value

```
h.addAttribute('CustomAttribute')
h.Attributes

ans =


# -------------------
# "Attributes" List
# -------------------
CustomAttribute = true
```

Add attribute using toolchain definition file

The `intel_tc.m` file from the “Adding a Custom Toolchain” example defines the following custom attributes:

```
tc.addAttribute('TransformPathsWithSpaces');
tc.addAttribute('RequiresCommandFile');
tc.addAttribute('RequiresBatchFile');
```

To see the property values from that example in the MATLAB Command Window, enter:

```
h = intel_tc;
h.Attributes


ans =


# -------------------
# "Attributes" List
# -------------------
RequiresBatchFile        = true
RequiresCommandFile      = true
TransformPathsWithSpaces = true
```

See Also

`addAttribute` | `getAttribute` | `getAttributes` | `isAttribute` | `removeAttribute`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addBuildConfiguration

Class: coder.make.ToolchainInfo
Package: coder.make

Add build configuration

Syntax

```
h.addBuildConfiguration(bldcfg_name)
h.addBuildConfiguration(bldcfg_name, bldcfg_desc)
h.addBuildConfiguration(bldcfg_handle)
```

Description

`h.addBuildConfiguration(bldcfg_name)` creates a `coder.make.BuildConfiguration` object, assigns the value of `bldcfg_name` to Name property of the object, and adds the object to coder.make.ToolchainInfo.BuildConfigurations on page 3-100.

`h.addBuildConfiguration(bldcfg_name, bldcfg_desc)` assigns the value of `bldcfg_desc` to Description property of the object.

`h.addBuildConfiguration(bldcfg_handle)` adds an existing build configuration object to coder.make.ToolchainInfo.BuildConfigurations on page 3-100. The build configuration must have a name that is unique within coder.make.ToolchainInfo.BuildConfigurations on page 3-100.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldcfg_name` — Build configuration name
character vector | string scalar

Build configuration name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldcfg_handle` — BuildConfiguration object handle
variable

Handle of `coder.make.BuildConfiguration` object

`bldcfg_desc` — Build configuration description
character vector | string scalar

Build configuration description, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

```
h.getBuildConfigurations
```

```
ans =

    'Faster Builds'
    'Faster Runs'
    'Debug'
```

```
bldcfg_handle = h.getBuildConfiguration('Debug')
```

```
bldcfg_handle =

############################################
# Build Configuration : Debug
# Description         : Default debug settings for compiling/linking code
############################################

ARFLAGS            = /nologo $(ARDEBUG)
CFLAGS             = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /Od $(CDEBUG)
CPPFLAGS           = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /Od $(CPPDEBUG)
DOWNLOAD_FLAGS     =
EXECUTE_FLAGS      =
LDFLAGS            = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) $(LDDEBUG)
MEX_CFLAGS         =
MEX_LDFLAGS        =
MAKE_FLAGS         = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS  = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE) $(LDDEBUG)
```

```
h.addBuildConfiguration('Debug2','Variant debugging configuration')
h.setBuildConfiguration('Debug2',bldcfg_handle)
h.getBuildConfigurations


ans =

    'Faster Builds'
    'Faster Runs'
    'Debug'
    'Debug2'
```

See Also

`coder.make.BuildConfiguration` | `coder.make.BuildItem` | `coder.make.BuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addBuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Add `BuildTool` object to `BuildTools`

Syntax

```
tool = h.addBuildTool(bldtl_name)
tool = h.addBuildTool(bldtl_name, bldtl_handle)
```

Description

`tool = h.addBuildTool(bldtl_name)` creates and adds a named `BuildTool` object to `coder.make.ToolchainInfo.BuildTools`.

`tool = h.addBuildTool(bldtl_name, bldtl_handle)` adds an existing `BuildTool` object to `coder.make.ToolchainInfo.BuildTools`. The `bldtl_name` argument overrides the `Name` property of the existing `BuildTool` object.

Tips

Refer to the “Example” on page 3-87 for `coder.make.BuildTool` for an example of how to create a `BuildTool` object.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Output Arguments

`tool` — BuildTool object handle
variable

Handle of `coder.make.BuildTool` object.

Examples

Refer to the coder.make.BuildTool “Example” on page 3-87 for a complete example of how to create a `addBuildTool`.

Create a build tool and specify its name

```
h.addBuildTool('ExampleBuildTool')
```

```
ans =

###########################################
# Build Tool: Build Tool
###########################################

Language              : 'C'
OptionsRegistry       : {}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<| |>OUTPUT<|'

# ---------
# Command
```

```
# ---------

# ------------
# Directives
# ------------
(none)

# -----------------
# File Extensions
# -----------------
(none)
```

See Also

`coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool`

Topics

"Adding a Custom Toolchain"
"About coder.make.ToolchainInfo"

addIntrinsicMacros

Class: coder.make.ToolchainInfo
Package: coder.make

Add intrinsic macro to `Macros`

Syntax

```
h.addIntrinsicMacros(intrnsc_macroname)
```

Description

`h.addIntrinsicMacros(intrnsc_macroname)` adds an intrinsic macro to `Macros`. The value of the intrinsic macro is defined by a build tool, not by ToolchainInfo or your MathWorks software.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`intrnsc_macronames` — Intrinsic macro name or names

character vector or cell array of character vectors | string or string array

Intrinsic macro name or names, specified as a character vector, cell array of character vectors, string, or string array.

Examples

```
h.addIntrinsicMacros('GCCROOT')
h.getMacro('GCCROOT')


ans  =

     []


h.removeIntrinsicMacros('GCCROOT')
h.getMacro('GCCROOT')
```

Tips

The value of intrinsic macros are intentionally not declared in `ToolchainInfo`. The value of the intrinsic macro is defined by the build tools in the toolchain, outside the scope of your MathWorks software.

During the software build process, your MathWorks software inserts intrinsic macros into a build artifact, such as a makefile, without altering their form. During the build process, the build artifact passes the intrinsic macros to the build tools in the toolchain. The build tools interpret the macros based on their own internal definitions.

The `validate` method does not validate the intrinsic macros.

Because intrinsic macros have undeclared values, they remain unchanged in the generated code, where they can be used and interpreted by the software build toolchain. In contrast, ordinary macros are replaced by their assigned values when you create them.

See Also

`addMacro` | `getMacro` | `removeMacro` | `setMacro` | `addIntrinsicMacros` | `removeIntrinsicMacros`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addMacro

Class: coder.make.ToolchainInfo
Package: coder.make

Add macro to `Macros`

Syntax

```
h.addMacro(macroname)
h.addMacro(macroname, macrovalue)
```

Description

`h.addMacro(macroname)` adds a macro to `coder.make.ToolchainInfo.Macros` without initializing the value of the Macro.

`h.addMacro(macroname, macrovalue)` adds a macro and initializes the value of the macro.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`macroname` — Name of macro

character vector | string scalar

Name of macro.

`macrovalue` — Value of macro

character vector | string scalar | cell

Value of the macro, specified as a character vector, string scalar, or cell array.

If the value contains MATLAB functions or other macros, ToolchainInfo on page 3-94 interprets the value of functions and macros.

Data Types: `cell` | `char` | `string`

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')
```

```
ans  =

C:\cygwin\bin\
```

```
h.removeMacro('CYGWIN')
```

Tips

Use `setMacro` to update the value of a macro in `coder.make.ToolchainInfo.Macros`.

See Also

`addMacro` | `getMacro` | `removeMacro` | `setMacro` | `addIntrinsicMacros` | `removeIntrinsicMacros`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addPostbuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Add postbuild tool to `PostbuildTools`

Syntax

```
h.addPostbuildTool(bldtl_name)
h.addPostbuildTool(bldtl_name, bldtl_handle)
```

Description

`h.addPostbuildTool(bldtl_name)` adds a `BuildTool` object to `PostbuildTools`.

`h.addPostbuildTool(bldtl_name, bldtl_handle)` adds a postbuild tool to `PostbuildTools` and assigns a `BuildTool` object to it.

Tips

Refer to the “Example” on page 3-87 for `coder.make.BuildTool` for an example of how to create a `BuildTool` object.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('postbuildtoolname');
h.addPostbuildTool('examplename',bt)

ans =

################################################
# Build Tool: postbuildtoolname
################################################

Language              : 'C'
OptionsRegistry       : {}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<| |>OUTPUT<|'

# ---------
# Command
# ---------

# ------------
# Directives
# ------------
(none)

# -----------------
# File Extensions
# -----------------
(none)
```

See Also

`addPostbuildTool` | `getPostbuildTool` | `removePostbuildTool` | `setPostbuildTool` | `addPostDownloadTool` | `addPostExecuteTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addPostDownloadTool

Class: coder.make.ToolchainInfo
Package: coder.make

Add post-download tool to `PostDownloadTool`

Syntax

```
h.addPostDownloadTool(bldtl_name,bldtl_handle)
```

Description

`h.addPostDownloadTool(bldtl_name,bldtl_handle)` adds a `BuildTool` object between the download tool and the execute tool specified by the `PostbuildTools` property.

Tips

Refer to the “Example” on page 3-87 for `coder.make.BuildTool` for an example of how to create a `BuildTool` object.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('toolname');
h.addPostDownloadTool('examplename',bt)

ans =

################################################
# Build Tool: toolname
################################################

Language              : 'C'
OptionsRegistry       : {}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<| |>OUTPUT<|'

# ---------
# Command
# ---------

# ------------
# Directives
# ------------
(none)

# -----------------
# File Extensions
# -----------------
(none)
```

References

“About coder.make.ToolchainInfo”

See Also

`addPostbuildTool` | `getPostbuildTool` | `removePostbuildTool` | `setPostbuildTool` | `addPostDownloadTool` | `addPostExecuteTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addPostExecuteTool

Class: coder.make.ToolchainInfo
Package: coder.make

Add post-execute tool to `PostbuildTools`

Syntax

```
h.addPostExecuteTool(name,bldtl_handle)
```

Description

`h.addPostExecuteTool(name,bldtl_handle)` adds a named build tool to `PostbuildTools` after the `Execute` tool.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`name` — Name of post execute tool
character vector | string scalar

Name of post execute tool, specified as a character vector or string scalar.

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

Refer to the coder.make.BuildTool “Example” on page 3-87 for an example of to create a `BuildTool`.

To use addPostExecuteTool, enter the following commands:

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('toolname');
h.addPostExecuteTool('ExampleName',bt)

ans =

################################################
# Build Tool: toolname
################################################

Language              : 'C'
OptionsRegistry       : {}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<| |>OUTPUT<|'

# ---------
# Command
# ---------

# ------------
# Directives
# ------------
(none)

# -----------------
# File Extensions
# -----------------
(none)
```

See Also

`addPostbuildTool` | `addPostDownloadTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

addPrebuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Add prebuild tool to `PrebuildTools`

Syntax

```
h.addPrebuildTool(bldtl_name)
h.addPrebuildTool(bldtl_name, bldtl_handle)
```

Description

`h.addPrebuildTool(bldtl_name)` creates a `BuildTool` object and adds it to the `PrebuildTools` property.

`h.addPrebuildTool(bldtl_name, bldtl_handle)` adds an existing `BuildTool` object to the `PrebuildTools` property.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

bldtl_name — Build tool name

character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

See Also

`addPrebuildTool` | `getPrebuildTool` | `removePrebuildTool` | `setPrebuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

cleanup

Class: coder.make.ToolchainInfo
Package: coder.make

Run cleanup commands

Syntax

```
h.cleanup
```

Description

`h.cleanup` runs cleanup commands after completing the software build process. First, it runs the commands specified by `coder.make.ToolchainInfo.ShellCleanup`, and then it runs the commands specified by `coder.make.ToolchainInfo.MATLABCleanup`.

The commands in `ShellCleanup` run as system calls to the standard input of the operating system on your host computer. These commands are similar to what you enter when you use the command line.

The commands in `MATLABCleanup` run in your MATLAB software.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Output Arguments

`success` — Indication whether cleanup completed

Indication whether cleanup completed (`0` = false, `1` = true), returned as a scalar.

Data Types: `double`

`report` — Information generated by the cleanup commands

Detailed information generated by the cleanup commands, returned as a character vector.

Data Types: `double`

Examples

```
[success,report] = h.cleanup

success  =

     1


report  =

     ''
```

See Also

`setup` | `validate`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getAttribute

Class: coder.make.ToolchainInfo
Package: coder.make

Get value of attribute

Syntax

```
att_value = h.getAttribute(att_name)
```

Description

`att_value = h.getAttribute(att_name)` returns the value of a specific attribute in `coder.make.ToolchainInfo.Attributes`.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`att_name` — Name of attribute
character vector | string scalar

Name of attribute, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`att_value` — Value of attribute
true (default)

Attribute value. Any data type.

Examples

```
h.Attribute
```

```
ans =


# -------------------
# "Attribute" List
# -------------------
(empty)
```

```
h.addAttribute('TransformPathsWithSpaces',false)
h.getAttribute('TransformPathsWithSpaces')
```

```
ans  =

     0
```

See Also

`addAttribute` | `getAttribute` | `getAttributes` | `isAttribute` | `removeAttribute`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getAttributes

Class: coder.make.ToolchainInfo
Package: coder.make

Get list of attribute names

Syntax

```
names = h.getAttributes
```

Description

`names = h.getAttributes` returns the list of attribute names in `coder.make.ToolchainInfo.Attributes`.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Output Arguments

names — List of names

cell array of character vectors

A list of the names, returned as a cell array.

Data Types: `cell`

Examples

```
h.addAttribute('FirstAttribute')
h.addAttribute('SecondAttribute')
h.addAttribute('ThirdAttribute')
names = h.getAttributes
```

```
names =

    'FirstAttribute'    'SecondAttribute'    'ThirdAttribute'
```

See Also

`addAttribute` | `getAttribute` | `getAttributes` | `isAttribute` | `removeAttribute`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getBuildConfiguration

Class: coder.make.ToolchainInfo
Package: coder.make

Get handle for build configuration object

Syntax

```
bldcfg_handle = h.getBuildConfiguration(bldcfg_name)
```

Description

`bldcfg_handle = h.getBuildConfiguration(bldcfg_name)` returns a handle for the specified coder.make.BuildConfig object.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldcfg_name` — Build configuration name
character vector | string scalar

Build configuration name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`bldcfg_handle` — BuildConfiguration object handle
variable

Handle of `coder.make.BuildConfiguration` object

Examples

```
bldcfg_handle = h.getBuildConfiguration('Debug')
```

```
bldcfg_handle =

############################################
# Build Configuration : Debug
# Description         : Default debug settings for compiling/linking code
############################################

ARFLAGS            = /nologo $(ARDEBUG)
CFLAGS             = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /Od $(CDEBUG)
CPPFLAGS           = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /Od $(CPPDEBUG)
DOWNLOAD_FLAGS     =
EXECUTE_FLAGS      =
LDFLAGS            = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) $(LDDEBUG)
MEX_CFLAGS         =
MEX_LDFLAGS        =
MAKE_FLAGS         = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS  = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE) $(LDDEBUG)
```

See Also

`getBuildConfiguration` | `removeBuildConfiguration` | `setBuildConfiguration` | `setBuildConfigurationOption`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getBuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Get `BuildTool` object

Syntax

```
bldtl_handle = h.getBuildTool(bldtl_name)
```

Description

`bldtl_handle = h.getBuildTool(bldtl_name)` returns the `BuildTool` object that has the specified name.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
bldtl_handle = h.getBuildTool('C Compiler')
```

```
bldtl_handle =

############################################
# Build Tool: Intel C Compiler
############################################

Language              : 'C'
OptionsRegistry       : {'C Compiler','CFLAGS'}
InputFileExtensions   : {'Source'}
OutputFileExtensions  : {'Object'}
DerivedFileExtensions : {'|>OBJ_EXT<|'}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
CC = icl
CC_PATH  =

# ------------
# Directives
# ------------
Debug              = -Zi
Include            =
IncludeSearchPath  = -I
OutputFlag         = -Fo
PreprocessorDefine = -D

# -----------------
```

```
# File Extensions
# -----------------
Header = .h
Object = .obj
Source = .c
```

See Also

`coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool`

Topics

"Adding a Custom Toolchain"
"About coder.make.ToolchainInfo"

getMacro

Class: coder.make.ToolchainInfo
Package: coder.make

Get value of macro

Syntax

```
value = h.getMacro(macroname)
```

Description

`value = h.getMacro(macroname)` returns the value of the specified macro.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`macroname` — Name of macro

character vector | string scalar

Name of macro.

Output Arguments

`macrovalue` — Value of macro

character vector | string scalar | cell

Value of the macro, specified as a character vector, string scalar, or cell array.

If the value contains MATLAB functions or other macros, ToolchainInfo on page 3-94 interprets the value of functions and macros.

Data Types: `cell` | `char` | `string`

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')
```

```
ans  =

C:\cygwin\bin\
```

```
h.removeMacro('CYGWIN')
```

See Also

`addMacro` | `getMacro` | `removeMacro` | `setMacro` | `addIntrinsicMacros` | `removeIntrinsicMacros`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getPostbuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Get postbuild `BuildTool` object

Syntax

```
bldtl_handle = h.getPostbuildTool(bldtl_name)
```

Description

`bldtl_handle = h.getPostbuildTool(bldtl_name)` gets the named `BuildTool` object from `PostbuildTool` and returns a handle to the object.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name

character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
h.getPostbuildTool('Download')
```

```
ans =

################################################
# Build Tool: Download
################################################

Language              : ''
OptionsRegistry       : {'Download','DOWNLOAD_FLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {coder.make.enum.BuildOutput.EXECUTABLE}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<|'

# ---------
# Command
# ---------
DOWNLOAD  =
DOWNLOAD_PATH  =

# ------------
# Directives
# ------------
(none)

# -----------------
# File Extensions
# -----------------
(none)
```

See Also

`addPostbuildTool` | `getPostbuildTool` | `removePostbuildTool` | `setPostbuildTool` | `addPostDownloadTool` | `addPostExecuteTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getPrebuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Get prebuild `BuildTool` object

Syntax

```
bldtl_handle = tc.getPrebuildTool(bldtl_name)
```

Description

`bldtl_handle = tc.getPrebuildTool(bldtl_name)` gets the named `BuildTool` object from `PrebuildTool` and assigns it to a handle.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
h.getPrebuildTool('Copy Tool')
```

```
ans =

################################################
# Build Tool: Copy Tool
################################################

Language              : ''
OptionsRegistry       : {'Copy','COPY_FLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {coder.make.enum.BuildOutput.EXECUTABLE}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<|'

# ---------
# Command
# ---------
COPY  =
COPY_PATH  =

# ------------
# Directives
# ------------
(none)

# -----------------
# File Extensions
# -----------------
(none)
```

See Also

`addPrebuildTool` | `getPrebuildTool` | `removePrebuildTool` | `setPrebuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

coder.make.ToolchainInfo.getSupportedLanguages

Class: coder.make.ToolchainInfo
Package: coder.make

Get list of supported languages

Syntax

```
lng_list = h.getSupportedLanguages
```

Description

`lng_list = h.getSupportedLanguages` returns the list of supported code generation languages for the current toolchain.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Output Arguments

lng_list — List of supported languages

cell

List of supported languages, returned as a cell.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

```
ans = h.getSupportedLanguages
```

```
ans =

    'Asm/C'    'Asm/C++'    'Asm/C/C++'    'C'    'C++'    'C/C++'
```

See Also

`coder.make.ToolchainInfo`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

isAttribute

Class: coder.make.ToolchainInfo
Package: coder.make

Determine if attribute exists

Syntax

```
truefalse = h.isAttribute(att_name)
```

Description

`truefalse = h.isAttribute(att_name)` returns a logical value that indicates whether the specified attribute is a member of `coder.make.ToolchainInfo.Attributes`.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`att_name` — Name of attribute

character vector | string scalar

Name of attribute, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`truefalse` — Logical value
boolean

Logical value: `0` = false, `1` = true, specified as a logical value.

Data Types: `logical`

Examples

```
h.addAttribute('FirstAttribute')
truefalse = h.isAttribute('FirstAttribute')
```

```
truefalse  =

     1
```

See Also

`addAttribute` | `getAttribute` | `getAttributes` | `isAttribute` | `removeAttribute`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removeAttribute

Class: coder.make.ToolchainInfo
Package: coder.make

Remove attribute

Syntax

```
h.removeAttribute(att_name)
```

Description

`h.removeAttribute(att_name)` removes the named attribute from `coder.make.ToolchainInfo.Attributes`.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

att_name — Name of attribute

character vector | string scalar

Name of attribute, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

```
h.addAttribute('FirstAttribute')
h.isAttribute('FirstAttribute')
```

```
ans  =

     1
```

```
h.removeAttribute('FirstAttribute')
h.isAttribute('FirstAttribute')
```

```
ans  =

     0
```

See Also

`addAttribute` | `getAttribute` | `getAttributes` | `isAttribute` | `removeAttribute`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removeBuildConfiguration

Class: coder.make.ToolchainInfo
Package: coder.make

Remove build configuration

Syntax

```
h.removeBuildConfiguration(bldcfg_name)
```

Description

`h.removeBuildConfiguration(bldcfg_name)` removes the specified build configuration object from `coder.make.ToolchainInfo.BuildConfiguration`.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

bldcfg_name — Build configuration name

character vector | string scalar

Build configuration name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

```
h.BuildConfigurations
```

```
ans =


# ----------------------------
# "BuildConfigurations" List
# ----------------------------
Debug                       =  <coder.make.BuildConfiguration>
ExampleName                       =  <coder.make.BuildConfiguration>
Faster Builds = <coder.make.BuildConfiguration>
Faster Runs   = <coder.make.BuildConfiguration>

h.removeBuildConfiguration('ExampleName')
h.BuildConfigurations


ans =


# ----------------------------
# "BuildConfigurations" List
# ----------------------------
Debug                       =  <coder.make.BuildConfiguration>
Faster Builds = <coder.make.BuildConfiguration>
Faster Runs   = <coder.make.BuildConfiguration>
```

See Also

`getBuildConfiguration` | `removeBuildConfiguration` | `setBuildConfiguration` | `setBuildConfigurationOption`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removeBuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Remove `BuildTool` object from `BuildTools`

Syntax

```
h.removeBuildTool(bldtl_name)
```

Description

`h.removeBuildTool(bldtl_name)` removes the named build tool from `BuildTools`.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

```
h.addBuildTool('ExampleBuildTool');
h.BuildTools
```

```
ans =


# -------------------
# "BuildTools" List
# -------------------
C Compiler       = <coder.make.BuildTool>
C++ Compiler     = <coder.make.BuildTool>
Archiver         = <coder.make.BuildTool>
Linker           = <coder.make.BuildTool>
MEX Tool         = <coder.make.BuildTool>
ExampleBuildTool = <coder.make.BuildTool>


h.removeBuildTool('ExampleBuildTool')
h.BuildTools


ans =


# -------------------
# "BuildTools" List
# -------------------
C Compiler       = <coder.make.BuildTool>
C++ Compiler     = <coder.make.BuildTool>
Archiver         = <coder.make.BuildTool>
Linker           = <coder.make.BuildTool>
MEX Tool         = <coder.make.BuildTool>
```

See Also

`coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removeIntrinsicMacros

Class: coder.make.ToolchainInfo
Package: coder.make

Remove intrinsic macro

Syntax

```
h.removeIntrinsicMacros(intrnsc_macronames)
```

Description

`h.removeIntrinsicMacros(intrnsc_macronames)` removes the named intrinsic macro from `Macros`.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`intrnsc_macronames` — Intrinsic macro name or names

character vector or cell array of character vectors | string or string array

Intrinsic macro name or names, specified as a character vector, cell array of character vectors, string, or string array.

Examples

```
h.addIntrinsicMacros('GCCROOT')
h.getMacro('GCCROOT')
```

```
ans  =

     []


h.removeIntrinsicMacros('GCCROOT')
h.getMacro('GCCROOT')
```

See Also

`addMacro` | `getMacro` | `removeMacro` | `setMacro` | `addIntrinsicMacros` | `removeIntrinsicMacros`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removeMacro

Class: coder.make.ToolchainInfo
Package: coder.make

Remove macro from `Macros`

Syntax

```
h.removeMacro(macroname)
```

Description

`h.removeMacro(macroname)` removes a macro from `coder.make.ToolchainInfo.Macros`.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

macroname — Name of macro

character vector | string scalar

Name of macro.

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')
```

```
ans  =

C:\cygwin\bin\


h.removeMacro('CYGWIN')
```

See Also

addMacro | getMacro | removeMacro | setMacro | addIntrinsicMacros | removeIntrinsicMacros

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removePostbuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Remove postbuild build tool

Syntax

```
h.removePostbuildTool(bldtl_name)
```

Description

`h.removePostbuildTool(bldtl_name)` removes the named build tool from `PostbuildTools`.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name

character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

```
h.addPostbuildTool('copier');
h.PostbuildTools
```

```
ans =


# -----------------------
# "PostbuildTools" List
# -----------------------
copier   = <coder.make.BuildTool>
Download = <coder.make.BuildTool>
Execute  = <coder.make.BuildTool>
```

```
h.removePostbuildTool('copier')
```

See Also

`addPostbuildTool` | `getPostbuildTool` | `removePostbuildTool` | `setPostbuildTool` | `addPostDownloadTool` | `addPostExecuteTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

removePrebuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Remove prebuild build tool

Syntax

```
h.removePrebuildTool(bldtl_name)
```

Description

`h.removePrebuildTool(bldtl_name)` removes the named build tool from `PrebuildTools`.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

bldtl_name — Build tool name

character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

If you have an example `coder.make.ToolchainInfo.PrebuildTools` object that contains a `BuildTool` object such as `copyFiles`:

```
h.PrebuildTools


ans =


# ----------------------
# "PrebuildTools" List
# ----------------------
copyFiles = <coder.make.BuildTool>
```

```
h.removePrebuildTool('copyFiles')
```

See Also

`addPrebuildTool` | `getPrebuildTool` | `removePrebuildTool` | `setPrebuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setBuildConfiguration

Class: coder.make.ToolchainInfo
Package: coder.make

Set value of specified build configuration

Syntax

```
h.setBuildConfiguration(bldcfg_name, bldcfg_handle)
```

Description

`h.setBuildConfiguration(bldcfg_name, bldcfg_handle)` assigns a build configuration object to a build configuration in `coder.make.ToolchainInfo.BuildConfigurations`.

Tips

Before you can use this method, add a build configuration to `BuildConfigurations` using `coder.make.ToolchainInfo.addBuildConfiguration` with a `bldcfg_name` argument.

Input Arguments

`h` — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldcfg_name` — Build configuration name

character vector | string scalar

Build configuration name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldcfg_handle` — BuildConfiguration object handle
variable

Handle of `coder.make.BuildConfiguration` object

Examples

```
h.getBuildConfigurations
```

```
ans =

    'Faster Builds'
    'Faster Runs'
    'Debug'
```

```
bldcfg_handle = h.getBuildConfiguration('Debug')
```

```
bldcfg_handle =

############################################
# Build Configuration : Debug
# Description         : Default debug settings for compiling/linking code
############################################

ARFLAGS            = /nologo $(ARDEBUG)
CFLAGS             = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /Od $(CDEBUG)
CPPFLAGS           = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /Od $(CPPDEBUG)
DOWNLOAD_FLAGS     =
EXECUTE_FLAGS      =
LDFLAGS            = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) $(LDDEBUG)
MEX_CFLAGS         =
MEX_LDFLAGS        =
MAKE_FLAGS         = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS  = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE) $(LDDEBUG)
```

```
h.addBuildConfiguration('Debug2','Variant debugging configuration')
h.setBuildConfiguration('Debug2',bldcfg_handle)
h.getBuildConfigurations
```

```
ans =

    'Faster Builds'
    'Faster Runs'
```

```
'Debug'
'Debug2'
```

See Also

`getBuildConfiguration` | `removeBuildConfiguration` | `setBuildConfiguration` | `setBuildConfigurationOption`

Topics

"Adding a Custom Toolchain"
"About coder.make.ToolchainInfo"

setBuildConfigurationOption

Class: coder.make.ToolchainInfo
Package: coder.make

Sets value of build tool options for build configuration

Syntax

```
h.setBuildConfigurationOption(buildconfignames, buildobjectname,
options)
```

Description

`h.setBuildConfigurationOption(buildconfignames, buildobjectname, options)` sets option values for the named `coder.make.BuildConfiguration` objects in `coder.make.ToolchainInfo.BuildConfigurations`.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`buildconfignames` — Build configuration names
character vector | string scalar

Build configuration name or `'all'`, specified as a character vector.

`buildobjectname` — BuildTool object name
character vector | string scalar

`BuildTool` object name, specified as a character vector.

`options` — Build configuration options
character vector | string scalar

Build configuration options, specified as a character vector.

Examples

To update a specific `BuildConfiguration` object or objects:

```
h = coder.make.ToolchainInfo
h.setBuildConfigurationOption('Faster Runs','C Compiler','-c -g')
```

To update all `BuildConfiguration` objects:

```
h = coder.make.ToolchainInfo
tc.setBuildConfigurationOption('all','C Compiler','-c -g')
```

See Also

`getBuildConfiguration` | `removeBuildConfiguration` | `setBuildConfiguration` | `setBuildConfigurationOption`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setBuilderApplication

Class: coder.make.ToolchainInfo
Package: coder.make

Update builder application to work on specific platform

Syntax

```
h.setBuilderApplication(platform)
```

Description

`h.setBuilderApplication(platform)` updates options in the `coder.make.BuildTool` object in `coder.make.ToolchainInfo.BuilderApplication` to work on a specific platform.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

platform — Development computer

character vector | string scalar

To specify development computer, use one of these values:

- `WIN32`
- `WIN64`
- `MACI64`

- `GLNXA64`

Data Types: `char` | `string`

Examples

The `intel_tc.m` file from “Adding a Custom Toolchain”, uses the following lines to update the `BuilderApplication` property:

```
% ------------------------------
% Builder
% ------------------------------

tc.setBuilderApplication(tc.Platform);
```

Tips

- You must use this method you if you plan to use the custom toolchain on a computer running Windows and the value of `coder.make.ToolchainInfo.BuildArtifact` is `gmake makefile`.

See Also

Topics

“Adding a Custom Toolchain”

setBuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Assign `BuildTool` object to named build tool in `BuildTools`

Syntax

```
h.setBuildTool(bldtl_name, bldtl_handle)
```

Description

`h.setBuildTool(bldtl_name, bldtl_handle)` assigns a `BuildTool` object to the named build tool in `coder.make.ToolchainInfo.BuildTools`.

Tips

Refer to the "Example" on page 3-87 for `coder.make.BuildTool` for an example of how to create a `BuildTool` object.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('examplename')
h.setBuildTool('Archiver',bt)
```

See Also

`coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setMacro

Class: coder.make.ToolchainInfo
Package: coder.make

Set value of macro

Syntax

```
h.setMacro(macroname, value)
```

Description

`h.setMacro(macroname, value)` sets the value of a macro.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

macroname — Name of macro

character vector | string scalar

Name of macro.

macrovalue — Value of macro

character vector | string scalar | cell

Value of the macro, specified as a character vector, string scalar, or cell array.

If the value contains MATLAB functions or other macros, ToolchainInfo on page 3-94 interprets the value of functions and macros.

Data Types: `cell` | `char` | `string`

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')
```

```
ans  =

C:\cygwin\bin\
```

```
h.removeMacro('CYGWIN')
```

See Also

`addMacro` | `getMacro` | `removeMacro` | `setMacro` | `addIntrinsicMacros` | `removeIntrinsicMacros`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setPostbuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Assign `BuildTool` object to `PostbuildTool` tool in `PostbuildTools`

Syntax

```
h.setPostbuildTool(bldtl_name, bldtl_handle)
```

Description

`h.setPostbuildTool(bldtl_name, bldtl_handle)` assigns a `BuildTool` object to the named build tool in coder.make.ToolchainInfo.PostbuildTools on page 3-110.

Input Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`bldtl_name` — Build tool name
character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`bldtl_handle` — BuildTool object handle
variable

`coder.make.BuildTool` object handle.

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('examplename')
h.addPostbuildTool('toolname')
h.setPostbuildTool('toolname',bt)
```

See Also

`addPostbuildTool` | `getPostbuildTool` | `removePostbuildTool` | `setPostbuildTool` | `addPostDownloadTool` | `addPostExecuteTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setPrebuildTool

Class: coder.make.ToolchainInfo
Package: coder.make

Assign `BuildTool` object to named `PrebuildTool` in `PrebuildTools`

Syntax

```
h.setPrebuildTool(bldtl_name, bldtl_handle)
```

Description

`h.setPrebuildTool(bldtl_name, bldtl_handle)` assigns a `BuildTool` object to the named build tool in coder.make.ToolchainInfo.PrebuildTools on page 3-111.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

bldtl_name — Build tool name

character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('examplename');
```

```
h.addPrebuildTool('toolname');
h.setPrebuildTool('toolname',bt)
```

See Also

`addPrebuildTool` | `getPrebuildTool` | `removePrebuildTool` | `setPrebuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

setup

Class: coder.make.ToolchainInfo
Package: coder.make

Invoke toolchain setup commands specified by MATLABSetup and ShellSetup

Syntax

```
h.setup
```

Description

`h.setup` runs setup commands before starting the software build process. First, it runs the commands specified by `coder.make.ToolchainInfo.MATLABSetup`, and then it runs the commands specified by `coder.make.ToolchainInfo.ShellSetup`.

The commands in `MATLABSetup` run in your MATLAB software.

The commands in `ShellSetup` run as system calls to the standard input of the operating system on your host computer. These commands are similar to what you enter when you use the command line.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Output Arguments

`success` — Response indicating whether setup completed
double

Response indicating whether setup completed (`0` = false, `1` = true), returned as a double.

`report` — Detailed information generated by setup commands
character vector

Detailed information generated by the setup commands, returned as a character vector.

Examples

```
[success,report] = h.setup


success  =

     1


report  =

     ''
```

See Also

`cleanup` | `validate`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

validate

Class: coder.make.ToolchainInfo
Package: coder.make

Validate toolchain

Syntax

```
h.validate
h.validate('setup','cleanup')
[success, report] = h.validate (___)
```

Description

`h.validate` validates the toolchain object and generates errors if any properties are incorrectly defined.

`h.validate('setup','cleanup')` evaluates the setup callbacks (`ShellSetup` and `MATLABSetup`) of the toolchain object before validation and evaluates the cleanup callbacks (`ShellCleanup` and `MATLABCleanup`) of the toolchain object after validation. The Configuration Parameters dialog box executes this version of validate when validating the toolchain.

`[success, report] = h.validate (___)` validates the toolchain object, generates errors if any properties are incorrectly defined, and returns optional output arguments.

Input Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`'setup'` — Setup argument for validate operation.
character vector | string scalar

Evaluates `setup` for the toolchain.

`'cleanup'` — Cleanup argument for validate operation.
character vector | string scalar

Evaluates `cleanup` for the toolchain.

Output Arguments

`success` — Response indicating whether validate passed
double

Response indicating whether validate passed, returned as a numeric value. If any of the property values the method checks are invalid, the method returns `0`. Otherwise, it returns `1`.

`report` — Information about which properties are invalid
character vector

Information about which properties are invalid. Only available when the method returns `0`.

Examples

Validate a toolchain before it has been installed

If you validate a default toolchain before all the build tools are specified, `validate` notifies you of the build tools that are not specified.

```
h = coder.make.ToolchainInfo;
[success,report] = h.validate
```

```
success  =

     1
```

```
report  =

Toolchain Validation Result: Passed

Validation report:

### Validation of build tool "C Compiler"
    Skipped. No "C Compiler" build tool is specified.

### Validation of build tool "C++ Compiler"
    Skipped. No "C++ Compiler" build tool is specified.

### Validation of build tool "Archiver"
    Skipped. No "Archiver" build tool is specified.

### Validation of build tool "Linker"
    Skipped. No "Linker" build tool is specified.

### Validation of build tool "MEX Tool"
Checking for existence of path: $(MATLAB_BIN)
    Passed.
Checking for tool command: mex
    Passed.

### Validation of build tool "Download"
    Skipped. No "Download" build tool is specified.

### Validation of build tool "Execute"
    Skipped. "Execute" build tool "$(PRODUCT)" cannot be validated.

### Validation of build tool "GMAKE Utility"
Checking for existence of path: %MATLAB%\bin\win64
    Passed.
Checking for tool command: gmake
    Passed.

### Checking for undeclared macros ...
    Passed.
```

Validate a toolchain before it has been installed

```
[success,report] = tc.validate
```

```
Error using ToolchainInfo.validate (line 270)
Validation error(s):
### Validating other build tools ...

Unable to locate build tool "Intel C Compiler": icl
 Unable to locate build tool "Intel C++ Compiler": icl
 Unable to locate build tool "Intel C/C++ Archiver": xilib
 Unable to locate build tool "Intel C/C++ Linker": xilink
 Unable to locate build tool "NMAKE Utility": nmake
```

See Also

`cleanup` | `setup`

Topics

“Adding a Custom Toolchain”
“Troubleshooting Custom Toolchain Validation”
“About coder.make.ToolchainInfo”

getDefaultToolchain

Get `DefaultToolchain` name from registry

Syntax

```
tc_name = coder.make.getDefaultToolchain()
```

Description

`tc_name = coder.make.getDefaultToolchain()` returns the name of the default host toolchain based on the toolchain indicated by `mex -setup`.

Output Arguments

`tc_name` — Toolchain name
character vector

Toolchain name, specified as a character vector.

Data Types: `char`

Examples

```
tc_name = coder.make.getDefaultToolchain()
```

```
tc_name =

  1×56 char array

Microsoft Visual C++ 2013 v12.0 | nmake (64-bit Windows)
```

See Also

`getToolchainInfoFromRegistry` | `coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getToolchainInfoFromRegistry

Get copy of selected `ToolchainInfo` object from registry

Syntax

```
h = coder.make.getToolchainInfoFromRegistry(tc_name)
```

Description

`h = coder.make.getToolchainInfoFromRegistry(tc_name)` returns a copy of the toolchain information object that has been registered with the specified name.

Input Arguments

`tc_name` — Toolchain name
character vector

Toolchain name, specified as a character vector.

Data Types: `char`

Output Arguments

`h` — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Examples

```
tc_name=coder.make.getDefaultToolchain();
h=coder.make.getToolchainInfoFromRegistry(tc_name)

h =

################################################################################
# Toolchain Name: Microsoft Visual C++ 2013 v12.0 | nmake (64-bit Windows)
# Supported Toolchain Version: 12.0
# Toolchain Specification Format Version: R2017a Prerelease
# Toolchain Specification Revision: 1.0
################################################################################

% try example to see the full toolchain info output in the command window
```

See Also

`getDefaultToolchain` | `coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool`

Topics

“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

getHardwareImplementation

Class: coder.BuildConfig
Package: coder

Get handle of copy of hardware implementation object

Syntax

```
hw = bldcfg.getHardwareImplementation()
```

Description

`hw = bldcfg.getHardwareImplementation()` returns the handle of a copy of the hardware implementation object.

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`hw`

Handle of copy of hardware implementation object.

See Also

`coder.HardwareImplementation`

getStdLibInfo

Class: coder.BuildConfig
Package: coder

Get standard library information

Syntax

```
[linkLibPath,linkLibExt,execLibExt,libPrefix]=
bldcfg.getStdLibInfo()
```

Description

`[linkLibPath,linkLibExt,execLibExt,libPrefix]= bldcfg.getStdLibInfo()` returns character vectors representing the:

- Standard MATLAB architecture-specific library path
- Platform-specific library file extension for use at link time
- Platform-specific library file extension for use at run time
- Standard architecture-specific library name prefix

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`linkLibPath`

Standard MATLAB architecture-specific library path specified as a character vector. The character vector can be empty.

`linkLibExt`

Platform-specific library file extension for use at link time, specified as a character vector. The value is one of `'.lib'`,`'.dylib'`,`'.so'`, `''`.

`execLibExt`

Platform-specific library file extension for use at run time, specified as a character vector. The value is one of `'.dll'`,`'.dylib'`,`'.so'`, `''`.

`linkPrefix`

Standard architecture-specific library name prefix, specified as a character vector. The character vector can be empty.

getTargetLang

Class: coder.BuildConfig
Package: coder

Get target code generation language

Syntax

```
lang = bldcfg.getTargetLang()
```

Description

`lang = bldcfg.getTargetLang()` returns a character vector containing the target code generation language.

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`lang`

A character vector containing the target code generation language. The value is `'C'` or `'C++'`.

getToolchainInfo

Class: coder.BuildConfig
Package: coder

Returns handle of copy of toolchain information object

Syntax

```
tc = bldcfg.getToolchainInfo()
```

Description

`tc = bldcfg.getToolchainInfo()` returns a handle of a copy of the toolchain information object.

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`tc`

Handle of copy of toolchain information object.

See Also

`coder.make.ToolchainInfo`

isCodeGenTarget

Class: coder.BuildConfig
Package: coder

Determine if build configuration represents specified target

Syntax

```
tf = bldcfg.isCodeGenTarget(target)
```

Description

`tf = bldcfg.isCodeGenTarget(target)` returns true (1) if the code generation target of the current build configuration represents the code generation target specified by `target`. Otherwise, it returns false (0).

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

`target`

Code generation target specified as a character vector or cell array of character vectors.

Specify	For code generation target
`'rtw'`	C/C++ dynamic Library, C/C++ static library, or C/C++ executable
`'sfun'`	S-function (Simulation)
`'mex'`	MEX-function

Specify `target` as a cell array of character vectors to test if the code generation target of the build configuration represents one of the targets specified in the cell array.

For example:

```
...
mytarget = {'sfun','mex'};
tf = bldcfg.isCodeGenTarget(mytarget);
...
```

tests whether the build context represents an S-function target or a MEX-function target.

Output Arguments

`tf`

The value is true (1) if the code generation target of the build configuration represents the code generation target specified by `target`. Otherwise, the value is false (0).

See Also

`coder.target`

isMatlabHostTarget

Class: coder.BuildConfig
Package: coder

Determine if hardware implementation object target is MATLAB host computer

Syntax

```
tf = bldcfg.isMatlabHostTarget()
```

Description

`tf = bldcfg.isMatlabHostTarget()` returns true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, it returns false (0).

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`tf`

Value is true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, the value is false (0).

See Also

`coder.HardwareImplementation`

isHeterogeneous

Class: coder.CellType
Package: coder

Determine whether cell array type represents a heterogeneous cell array

Syntax

```
tf = isHeterogeneous(t)
```

Description

`tf = isHeterogeneous(t)` returns `true` if the `coder.CellType` object `t` is heterogeneous. Otherwise, it returns `false`.

Examples

Determine Whether Cell Array Type Is Heterogeneous

Create a `coder.CellType` object for a cell array whose elements have different classes.

```
t = coder.typeof({'a', 1})
```

```
t =

coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 char
      f1: 1x1 double
```

Determine whether the `coder.CellType` object represents a heterogeneous cell array.

```
isHeterogeneous(t)
```

```
ans =

     1
```

Test for Heterogeneous Cell Array Type Before Executing Code

Write a function `assign_name`. If the input type `t` is heterogeneous, the function returns a copy of `t`. The copy specifies the name for the structure type that represents the cell array type in the generated code.

```
function ts = assign_name(t, str_name)
assert(isHeterogeneous(t));
ts = coder.cstructname(t, str_name);
disp ts
end
```

Create a homogeneous type `tc`.

```
tc = coder.typeof({1 2 3});
```

Pass `tc` to `make_varsize`.

```
tc1 = assign_name(tc, 'myname')
```

The assertions fails because `tc` is not heterogeneous.

Create a heterogeneous type `tc`.

```
tc = coder.typeof({'a' 1});
```

Pass `tc` to `make_varsize`.

```
tc1 = assign_name(tc, 'myname')
```

```
tc1 =

coder.CellType
   1x2 heterogeneous cell myname
```

```
f0: 1x1 char
f1: 1x1 double
```

Tips

- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

  The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

See Also

`coder.newtype` | `coder.typeof`

Topics

“Code Generation for Cell Arrays”
“Specify Cell Array Inputs at the Command Line”

Introduced in R2015b

isHomogeneous

Class: coder.CellType
Package: coder

Determine whether cell array type represents a homogeneous cell array

Syntax

```
tf = isHomogeneous(t)
```

Description

`tf = isHomogeneous(t)` returns `true` if the `coder.CellType` object `t` represents a homogeneous cell array. Otherwise, it returns `false`.

Examples

Determine Whether Cell Array Type Is Homogeneous.

Create a `coder.CellType` object for a cell array whose elements have the same class and size.

```
t = coder.typeof({1 2 3})
```

```
t =

coder.CellType
   1x3 homogeneous cell
      base: 1x1 double
```

Determine whether the `coder.CellType` object represents a homogeneous cell array.

```
isHomogeneous(t)
```

```
ans =

     1
```

Test for a Homogeneous Cell Array Type Before Executing Code

Write a function `make_varsize`. If the input type `t` is homogeneous, the function returns a variable-size copy of `t`.

```
function c = make_varsize(t, n)
assert(isHomogeneous(t));
c = coder.typeof(t, [n n], [1 1]);
end
```

Create a heterogeneous type `tc`.

```
tc = coder.typeof({'a', 1});
```

Pass `tc` to `make_varsize`.

```
tc1 = make_varsize(tc, 5)
```

The assertion fails because `tc` is heterogeneous.

Create a homogeneous type `tc`.

```
tc = coder.typeof({1 2 3});
```

Pass `tc` to `make_varsize`.

```
tc1 = make_varsize(tc, 5)
```

```
tc1 =

coder.CellType
   :5x:5 homogeneous cell
      base: 1x1 double
```

Tips

- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size,

`coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

See Also

`coder.newtype` | `coder.typeof`

Topics

“Code Generation for Cell Arrays”
“Specify Cell Array Inputs at the Command Line”

Introduced in R2015b

makeHeterogeneous

Class: coder.CellType
Package: coder

Make a heterogeneous copy of a cell array type

Syntax

```
newt = makeHeterogeneous(t)
t = makeHeterogeneous(t)
```

Description

`newt = makeHeterogeneous(t)` creates a `coder.CellType` object for a heterogeneous cell array from the `coder.CellType` object `t`. `t` cannot represent a variable-size cell array.

The classification as heterogeneous is permanent. You cannot later create a homogeneous `coder.CellType` object from `newt`.

`t = makeHeterogeneous(t)` creates a heterogeneous `coder.CellType` object from `t` and replaces `t` with the new object.

Examples

Replace a Homogeneous Cell Array Type with a Heterogeneous Cell Array Type

Create a cell array type `t` whose elements have the same class and size.

```
t = coder.typeof({1 2 3})
```

```
t =

coder.CellType
```

```
   1x3 homogeneous cell
      base: 1x1 double
```

The cell array type is homogeneous.

Replace `t` with a cell array type for a heterogeneous cell array.

```
t = makeHeterogeneous(t)
```

```
t =

coder.CellType
   1×3 locked heterogeneous cell
      f1: 1×1 double
      f2: 1×1 double
      f3: 1×1 doublee
```

The cell array type is heterogeneous. The elements have the size and class of the original homogeneous cell array type.

Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods.

See Also

`coder.newtype` | `coder.typeof`

Topics

“Code Generation for Cell Arrays”
“Specify Cell Array Inputs at the Command Line”

Introduced in R2015b

makeHomogeneous

Class: coder.CellType
Package: coder

Create a homogeneous copy of a cell array type

Syntax

```
newt = makeHomogeneous(t)
t = makeHomogeneous(t)
```

Description

`newt = makeHomogeneous(t)` creates a `coder.CellType` object for a homogeneous cell array `newt` from the `coder.CellType` object `t`.

To create `newt`, the `makeHomogeneous` method must determine a size and class that represent all elements of `t`:

- If the elements of `t` have the same class, but different sizes, the elements of `newt` are variable size with upper bounds that accommodate the elements of `t`.
- If the elements of `t` have different classes, for example, `char` and `double`, the `makeHomogeneous` method cannot create a `coder.CellType` object for a homogeneous cell array.

If you use `coder.cstructname` to specify a name for the structure type that represents `t` in the generated code, you cannot create a homogeneous `coder.CellType` object from `t`.

The classification as homogeneous is permanent. You cannot later create a heterogeneous `coder.CellType` object from `newt`.

`t = makeHomogeneous(t)` creates a homogeneous `coder.CellType` object from `t` and replaces `t` with the new object.

Examples

Replace a Heterogeneous Cell Array Type with a Homogeneous Cell Array Type

Create a cell array type `t` whose elements have the same class, but different sizes.

```
t = coder.typeof({1 [2 3]})
```

```
t =

coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 double
      f1: 1x2 double
```

The cell array type is heterogeneous.

Replace `t` with a cell array type for a homogeneous cell array.

```
t = makeHomogeneous(t)
```

```
t =

coder.CellType
   1×2 locked homogeneous cell
      base: 1×:2 double
```

The new cell array type is homogeneous.

Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the

type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods.

See Also

`coder.cstructname` | `coder.newtype` | `coder.typeof`

Topics

“Code Generation for Cell Arrays”
“Specify Cell Array Inputs at the Command Line”

Introduced in R2015b

coder.ExternalDependency.getDescriptiveName

Class: coder.ExternalDependency
Package: coder

Return descriptive name for external dependency

Syntax

```
extname = coder.ExternalDependency.getDescriptiveName(bldcfg)
```

Description

`extname = coder.ExternalDependency.getDescriptiveName(bldcfg)` returns the name that you want to associate with an "external dependency" on page 2-403. The code generator uses the external dependency name for error messages.

Input Arguments

`bldcfg`

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-403

You can use this information when you want to return different names based on the build context.

Output Arguments

`extname`

External dependency name returned as a character vector.

Examples

Return external dependency name

Define a method that always returns the same name.

```
function myextname = getDescriptiveName(~)
   myextname = 'MyLibrary'
end
```

Return external library name based on the code generation target

Define a method that uses the build context to determine the name.

```
function myextname = getDescriptiveName(context)
    if context.isMatlabHostTarget()
        myextname = 'MyLibary_MatlabHost';
    else
        myextname = 'MyLibrary_Local';
    end
end
```

Definitions

external dependency

External code interface represented by a class derived from a `coder.ExternalDependency` class. The external code can be a library, object files, or C/C++ source.

build context

Information used by the build process including:

- Target language
- Code generation target

- Target hardware
- Build toolchain

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo`

Topics

“Develop Interface for External C/C++ Code”
“Build Process Customization”
“Integrate External/Custom Code”

coder.ExternalDependency.isSupportedContext

Class: coder.ExternalDependency
Package: coder

Determine if build context supports external dependency

Syntax

```
tf = coder.ExternalDependency.isSupportedContext(bldcfg)
```

Description

`tf = coder.ExternalDependency.isSupportedContext(bldcfg)` returns true (1) if you can use the "external dependency" on page 2-406 in the current "build context" on page 2-406 . You must provide this method in the class definition for a class that derives from `coder.ExternalDependency`.

If you cannot use the "external dependency" on page 2-406 in the current "build context" on page 2-406, display an error message and stop code generation. The error message must describe why you cannot use the external dependency in this build context. If the method returns false (0), the code generator uses a default error message. The default error message uses the name returned by the `getDescriptiveName` method of the `coder.ExternalDependency` class.

Use `coder.BuildConfig` methods to determine if you can use the external dependency in the current build context.

Input Arguments

`bldcfg`

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-406.

Output Arguments

`tf`

Value is true (1) if the build context supports the external dependency.

Examples

Report error when build context does not support external library

This method returns true(1) if the code generation target is a MATLAB host target. Otherwise, the method reports an error and stops code generation.

Write `isSupportedContext` method.

```
function tf = isSupportedContext(ctx)
    if  ctx.isMatlabHostTarget()
        tf = true;
    else
        error('adder library not available for this target');
    end
end
```

Definitions

external dependency

External code interface represented by a class derived from `coder.ExternalDependency` class. The external code can be a library, object file, or C/C++ source.

build context

Information used by the build process including:

- Target language

- Code generation target
- Target hardware
- Build toolchain

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo`

Topics

“Develop Interface for External C/C++ Code”
“Build Process Customization”
“Integrate External/Custom Code”

coder.ExternalDependency.updateBuildInfo

Class: coder.ExternalDependency
Package: coder

Update build information

Syntax

```
coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)
```

Description

`coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)` updates the build information object whose handle is `buildInfo`. After code generation, the build information object has standard information. Use this method to provide additional information required to link to external code. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-409.

You must implement this method in a subclass of `coder.ExternalDependency`. For an example, see `coder.ExternalDependency`.

Input Arguments

`buildInfo`

Handle of build information object.

`bldcfg`

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-409.

Limitations

- The build information method `AddIncludeFiles` has no effect in a `coder.ExternalDependency` `updateBuildInfo` method.

Definitions

build context

Information used by the build process including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo`

Topics

“Develop Interface for External C/C++ Code”
“Build Process Customization”
“Integrate External/Custom Code”

addDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Add design range specification to parameter

Syntax

```
addDesignRangeSpecification(fcnName,paramName,designMin, designMax)
```

Description

`addDesignRangeSpecification(fcnName,paramName,designMin, designMax)` specifies the minimum and maximum values allowed for the parameter, `paramName`, in function, `fcnName`. The fixed-point conversion process uses this design range information to derive ranges for downstream variables in the code.

Input Arguments

`fcnName` — Function name
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

`designMin` — Minimum value allowed for this parameter
scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

`designMax` — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

Add a Design Range Specification

```
% Set up the fixed-point configuration object
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
fixptcfg.ComputeDerivedRanges = true;

%Set up C code configuration object
cfg = coder.config('lib');
% Derive ranges  and generate fixed-point C code
codegen -config cfg -float2fixed fixptcfg dti -report
```

See Also

`codegen` | `coder.FixptConfig` |
`coder.FixptConfig.clearDesignRangeSpecifications` |
`coder.FixptConfig.getDesignRangeSpecification` |
`coder.FixptConfig.hasDesignRangeSpecification` |
`coder.FixptConfig.removeDesignRangeSpecification`

addFunctionReplacement

Class: coder.FixptConfig
Package: coder

Replace floating-point function with fixed-point function during fixed-point conversion

Syntax

```
addFunctionReplacement(floatFn,fixedFn)
```

Description

`addFunctionReplacement(floatFn,fixedFn)` specifies a function replacement in a `coder.FixptConfig` object. During floating-point to fixed-point conversion, the conversion process replaces the specified floating-point function with the specified fixed-point function. The fixed-point function must be in the same folder as the floating-point function or on the MATLAB path.

Input Arguments

`floatFn` — Name of floating-point function

`''` (default) | string

Name of floating-point function, specified as a string.

`fixedFn` — Name of fixed-point function

`''` (default) | string

Name of fixed-point function, specified as a string.

Examples

Specify Function Replacement in Fixed-Point Conversion Configuration Object

Suppose that:

- The function `myfunc` calls a local function `myadd`.
- The test function `mytest` calls `myfunc`.
- You want to replace calls to `myadd` with the fixed-point function `fi_myadd`.

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `mytest`.

```
fixptcfg.TestBenchName = 'mytest';
```

Specify that the floating-point function, `myadd`, should be replaced with the fixed-point function, `fi_myadd`.

```
fixptcfg.addFunctionReplacement('myadd', 'fi_myadd');
```

Create a code generation configuration object to generate a standalone C static library.

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

Convert the floating-point MATLAB function, `myfunc`, to fixed-point, and generate C code.

```
codegen -float2fixed fixptcfg -config cfg myfunc
```

When you generate code, the code generator replaces instances of `myadd` with `fi_myadd` during floating-point to fixed-point conversion.

See Also

`codegen` | `coder.FixptConfig` | `coder.config`

clearDesignRangeSpecifications

Class: coder.FixptConfig
Package: coder

Clear all design range specifications

Syntax

```
clearDesignRangeSpecifications()
```

Description

`clearDesignRangeSpecifications()` clears all design range specifications.

Examples

Clear a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

See Also

`codegen` | `coder.FixptConfig` |
`coder.FixptConfig.addDesignRangeSpecification` |
`coder.FixptConfig.getDesignRangeSpecification` |

```
coder.FixptConfig.hasDesignRangeSpecification|
coder.FixptConfig.removeDesignRangeSpecification
```

getDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Get design range specifications for parameter

Syntax

```
[designMin, designMax] = getDesignRangeSpecification(fcnName,
paramName)
```

Description

`[designMin, designMax] = getDesignRangeSpecification(fcnName, paramName)` gets the minimum and maximum values specified for the parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Function name
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

Output Arguments

designMin — Minimum value allowed for this parameter
scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

designMax — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

Get Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Get the  design range for the 'dti' function parameter 'u_in'
[designMin, designMax] = cfg.getDesignRangeSpecification('dti','u_in')
```

```
designMin =

    -1


designMax =

     1
```

See Also

`codegen` | `coder.FixptConfig` | `coder.FixptConfig.addDesignRangeSpecification` |

```
coder.FixptConfig.clearDesignRangeSpecifications|
coder.FixptConfig.hasDesignRangeSpecification|
coder.FixptConfig.removeDesignRangeSpecification
```

hasDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Determine whether parameter has design range

Syntax

```
hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)
```

Description

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)` returns true if the parameter, `param_name` in function, `fcn`, has a design range specified.

Input Arguments

`fcnName` — Name of function

string

Function name, specified as a string.

Example: 'dti'

Data Types: `char`

`paramName` — Parameter name

string

Parameter name, specified as a string.

Example: 'dti'

Data Types: `char`

Output Arguments

hasDesignRange — Parameter has design range
true | false

Parameter has design range, returned as a boolean.

Data Types: `logical`

Examples

Verify That a Parameter Has a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

```
hasDesignRanges =

     1
```

See Also

`codegen` | `coder.FixptConfig` | `coder.FixptConfig.addDesignRangeSpecification` | `coder.FixptConfig.clearDesignRangeSpecifications` | `coder.FixptConfig.getDesignRangeSpecification` | `coder.FixptConfig.removeDesignRangeSpecification`

removeDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Remove design range specification from parameter

Syntax

```
removeDesignRangeSpecification(fcnName,paramName)
```

Description

`removeDesignRangeSpecification(fcnName,paramName)` removes the design range information specified for parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Name of function
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

Examples

Remove Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now clear the design ranges and verify that
% hasDesignRangeSpecification returns false
cfg.removeDesignRangeSpecification('dti', 'u_in')
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

See Also

codegen | coder.FixptConfig |
coder.FixptConfig.addDesignRangeSpecification |
coder.FixptConfig.clearDesignRangeSpecifications |
coder.FixptConfig.getDesignRangeSpecification |
coder.FixptConfig.hasDesignRangeSpecification

addApproximation

Replace floating-point function with lookup table during fixed-point conversion

Syntax

```
addApproximation(approximationObject)
```

Description

`addApproximation(approximationObject)` specifies a lookup table replacement in a `coder.FixptConfig` object. During floating-point to fixed-point conversion, the conversion process generates a lookup table approximation for the function specified in the `approximationObject`.

Input Arguments

`approximationObject` — Function replacement configuration object
`coder.mathfcngenerator.LookupTable` configuration object

Function replacement configuration object. Use the `coder.FixptConfig` configuration object `addApproximation` method to associate this configuration object with a `coder.FixptConfig` object. Then use the `codegen` function `-float2fixed` option with `coder.FixptConfig` to convert floating-point MATLAB code to fixed-point code.

Examples

Replace log function with an optimized lookup table replacement

Create a function replacement configuration object that specifies to replace the log function with an optimized lookup table.

```
logAppx = coder.approximation('Function','log','OptimizeLUTSize',...
          true,'InputRange',[0.1,1000],'InterpolationDegree',1,...
```

```
        'ErrorThreshold',1e-3,...
        'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Create a fixed-point configuration object and associate the function replacement configuration object with it.

```
fixptcfg = coder.config('fixpt');
fixptcfg.addApproximation(logAppx);
```

You can now generate fixed-point code using the `codegen` function.

See Also

`codegen` | `coder.FixptConfig` | `coder.config`

Topics

“Replace the exp Function with a Lookup Table”
“Replace a Custom Function with a Lookup Table”
“Replacing Functions Using Lookup Table Approximations”

addFunctionReplacement

Class: coder.SingleConfig
Package: coder

Replace double-precision function with single-precision function during single-precision conversion

Syntax

```
addFunctionReplacement(doubleFn,singleFn)
```

Description

`addFunctionReplacement(doubleFn,singleFn)` specifies a function replacement in a `coder.SingleConfig` object. During double-precision to single-precision conversion, the conversion process replaces the specified double-precision function with the specified single-precision function. The single-precision function must be in the same folder as the double-precision function or on the MATLAB path. It is a best practice to provide unique names to local functions that a replacement function calls. If a replacement function calls a local function, do not give that local function the same name as a local function in a different replacement function file.

Input Arguments

`doubleFn` — Name of double-precision function

`''` (default) | string

Name of double-precision function, specified as a string.

`singleFn` — Name of single-precision function

`''` (default) | string

Name of single-precision function, specified as a string.

Examples

Specify Function Replacement in Single-Precision Conversion Configuration Object

Suppose that:

- The function `myfunc` calls a local function `myadd`.
- The test function `mytest` calls `myfunc`.
- You want to replace calls to `myadd` with the single-precision function `single_myadd`.

Create a `coder.SingleConfig` object, `scfg`, with default settings.

```
scfg = coder.config('single');
```

Set the test file name. In this example, the test file function name is `mytest`.

```
scfg.TestBenchName = 'mytest';
```

Specify that you want to replace the double-precision function, `myadd`, with the single-precision function, `single_myadd`.

```
scfg.addFunctionReplacement('myadd', 'single_myadd');
```

Convert the double-precision MATLAB function, `myfunc`, to a single-precision MATLAB function.

```
codegen -double2single scfg myfunc
```

The double-precision to single-precision conversion replaces instances of `myadd` with `single_myadd`.

See Also

`codegen` | `coder.config`

Introduced in R2015b

coder.LAPACKCallback.getHeaderFilename

Class: coder.LAPACKCallback
Package: coder

Return file name of LAPACKE header file

Syntax

```
coder.LAPACKCallback.getHeaderFilename()
```

Description

`coder.LAPACKCallback.getHeaderFilename()` returns the file name of the LAPACKE header file that defines the C interface to a specific LAPACK library.

`coder.LAPACKCallback` is an abstract class for defining a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file to use for LAPACK calls in code generated from MATLAB code. At code generation time, if you specify a LAPACK callback class, for certain linear algebra function calls, the code generator produces LAPACK calls in standalone code.

The code generator uses the LAPACKE header file name to generate a `#include` statement.

Examples

Return LAPACKE Header File Name

This example shows how to write a `getHeaderFilename` method to return the name of the LAPACKE header file.

In a class that derives from `coder.LAPACKCallback`, write a method `getHeaderFilename` that returns the name of the LAPACKE header file as a character

vector. For example, in this class definition, `getHeaderFilename` returns `'mylapacke_custom.h'`.

```
classdef useMyLAPACK < coder.LAPACKCallback
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
        function updateBuildInfo(buildInfo, buildctx)
            buildInfo.addIncludePaths(fullfile(pwd,'include'));
            libName = 'mylapack';
            libPath = fullfile(pwd,'lib');
            [~,linkLibExt] = buildctx.getStdLibInfo();
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
                '', true, true);
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
        end
    end
end
```

See Also

Topics

“Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”
“Speed Up Linear Algebra in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.netlib.org/lapack

coder.LAPACKCallback.updateBuildInfo

Class: coder.LAPACKCallback
Package: coder

Update build information for linking to a specific LAPACK library

Syntax

```
coder.LAPACKCallback.updateBuildInfo(buildInfo, buildctx)
```

Description

`coder.LAPACKCallback.updateBuildInfo(buildInfo, buildctx)` updates the build information object `buildInfo` with the information required for the build process to link to a specific LAPACK library.

`coder.LAPACKCallback` is an abstract class for defining a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file to use for LAPACK calls in code generated from MATLAB code. At code generation time, if you specify a LAPACK callback class, for certain linear algebra function calls, the code generator produces LAPACK calls in standalone code.

Input Arguments

`buildInfo`

Build information object. After code generation, this object contains standard project, build option, and dependency information. In the `updateBuildInfo` method, to add the information for linking to the LAPACK library, use build information methods.

`buildctx`

`coder.BuildConfig` object. Use the `coder.BuildConfig` `getStdLibInfo` method to get the platform-specific file extension to use at link time.

Examples

Link to a Specific LAPACK Library

This example shows how to write an `updateBuildInfo` method to update the build information object with the information required to link to a specific LAPACK library.

In a class that derives from `coder.LAPACKCallback`, write a method `updateBuildInfo`. Use this example LAPACK callback class as a template.

```
classdef useMyLAPACK < coder.LAPACKCallback
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
        function updateBuildInfo(buildInfo, buildctx)
            buildInfo.addIncludePaths(fullfile(pwd,'include'));
            libName = 'mylapack';
            libPath = fullfile(pwd,'lib');
            [~,linkLibExt] = buildctx.getStdLibInfo();
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
                '', true, true);
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
        end
    end
end
```

Replace `mylapack` with the name of your LAPACK library. Modify the include and library paths as necessary.

To update the build information with the location of the header files, use the build information `addIncludePaths` method.

To access the platform-specific library extension, use the `coder.BuildConfig` `getStdLibInfo` method.

To update the build information with the name and location of your LAPACK library, use the build information `addlinkObjects` method.

If your compiler supports only complex data types that are represented as structures, include these lines.

```
buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
```

See Also

`coder.BuildConfig` | `coder.ExternalDependency`

Topics

“Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”
“Speed Up Linear Algebra in Code Generated from a MATLAB Function Block” (Simulink Coder)
“Build Information Methods”

External Websites

www.netlib.org/lapack

coder.BLASCallback.updateBuildInfo

Class: coder.BLASCallback
Package: coder

Update build information for linking to a specific BLAS library

Syntax

```
coder.BLASCallback.updateBuildInfo(buildInfo, buildctx)
```

Description

`coder.BLASCallback.updateBuildInfo(buildInfo, buildctx)` updates the build information object `buildInfo` with the information required for the build process to link to a specific BLAS library.

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. At code generation time, if you specify a BLAS callback class, for certain vector and matrix function calls, the code generator produces BLAS calls in standalone code.

`updateBuildInfo` is an abstract method. You must implement it in the definition of your callback class that derives from `coder.BLASCallback`.

Input Arguments

`buildInfo` — Build information object

value object (default)

After code generation, this object contains standard project, build option, and dependency information. In the `updateBuildInfo` method, to add the information for linking to the BLAS library, use build information methods.

`buildctx` — `coder.BuildConfig` object
value object (default)

Use the `coder.BuildConfig` methods to access the build context settings such as target language and platform-specific library file extensions.

Attributes

`Abstract`	`true`
`Static`	`true`

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

Link to the Intel MKL BLAS Library

This example shows how to write an `updateBuildInfo` method to update the build information object with the information required to link to a specific BLAS library.

In a class that derives from `coder.BLASCallback`, write a method `updateBuildInfo`. This example is an implementation of the callback class `mklcallback` for integration with the Intel MKL BLAS library on a Windows platform. Use this example BLAS callback class as a template.

```
classdef mklcallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, ~)
            libPath = fullfile(pwd,'mkl','WIN','lib','intel64');
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libs = {'mkl_intel_ilp64.lib' 'mkl_intel_thread.lib' 'mkl_core.lib'};
            buildInfo.addLinkObjects(libs, libPath, libPriority, libPreCompiled, libLi
            buildInfo.addLinkObjects('libiomp5md.lib',fullfile(matlabroot,'bin','win64
                libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(fullfile(pwd,'mkl','WIN','include'));
            buildInfo.addDefines('-DMKL_ILP64');
```

```
        end
        function headerName = getHeaderFilename()
            headerName = 'mkl_cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
            intTypeName = 'MKL_INT';
        end
    end
end
```

To update the build information with the name and location of your BLAS library, use the build information `addlinkObjects` method. If you use the Intel MKL BLAS library, use the link line advisor to see which libraries and compiler options are recommended for your use case.

To update the build information with the location of the header files, use the build information `addIncludePaths` method.

To add preprocessor macro definitions to the build information in `updateBuildInfo`, use the build information `addDefines` method.

To specify the compiler options in `updateBuildInfo`, use the build information `addCompileFlags` method.

To specify the linker options, use the build information `addLinkFlags` method.

See Also

`coder.BuildConfig` | `coder.ExternalDependency`

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)
“Build Process Customization”

External Websites

http://www.netlib.org/blas/
http://www.openblas.net/
https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines

https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

Introduced in R2018b

coder.BLASCallback.getHeaderFilename

Class: coder.BLASCallback
Package: coder

Return the file name of CBLAS header file

Syntax

```
headerName = coder.BLASCallback.getHeaderFilename()
```

Description

`headerName = coder.BLASCallback.getHeaderFilename()` returns the name of the CBLAS header file that defines the C interface to a specific BLAS library.

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. At code generation time, if you specify a BLAS callback class, for certain vector and matrix function calls, the code generator produces BLAS calls in standalone code.

`getHeaderFilename` is an abstract method. You must implement it in the definition of your callback class that derives from `coder.BLASCallback`. The code generator uses the CBLAS header file name returned by `getHeaderFilename` to produce a `#include` statement in the generated code.

Output Arguments

`headerName` — CBLAS header file name
character vector

Character vector that specifies the name of the CBLAS header file that defines the C interface to a specific BLAS library.

Attributes

Abstract	true
Static	true

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

Return CBLAS Header File Name

This example shows how to write a `getHeaderFilename` method to return the name of the CBLAS header file.

In a class that derives from `coder.BLASCallback`, write a method `getHeaderFilename` that returns the name of the CBLAS header file as a character vector. This example is an implementation of the callback class `mklcallback` for integration with the Intel MKL BLAS library on a Windows platform. In this class, `getHeaderFilename` returns `'mkl_cblas.h'`, which is the CBLAS header file for the Intel MKL BLAS library.

```
classdef mklcallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, ~)
            libPath = fullfile(pwd,'mkl','WIN','lib','intel64');
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libs = {'mkl_intel_ilp64.lib' 'mkl_intel_thread.lib' 'mkl_core.lib'};
            buildInfo.addLinkObjects(libs, libPath, libPriority, libPreCompiled, libLi
            buildInfo.addLinkObjects('libiomp5md.lib',fullfile(matlabroot,'bin','win64
                libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(fullfile(pwd,'mkl','WIN','include'));
            buildInfo.addDefines('-DMKL_ILP64');
        end
        function headerName = getHeaderFilename()
            headerName = 'mkl_cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
```

```
            intTypeName = 'MKL_INT';
        end
    end
end
```

If you are using a different BLAS library, replace `'mkl_cblas.h'` with the name of your CBLAS header file.

See Also

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

http://www.netlib.org/blas/
https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines

Introduced in R2018b

coder.BLASCallback.getBLASIntTypeName

Class: coder.BLASCallback
Package: coder

Return name of integer data type used by CBLAS interface

Syntax

```
intTypeName = coder.BLASCallback.getBLASIntTypeName()
```

Description

`intTypeName = coder.BLASCallback.getBLASIntTypeName()` returns the name of the integer data type that is used by the CBLAS interface to a specific BLAS library.

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. At code generation time, if you specify a BLAS callback class, for certain vector and matrix function calls, the code generator produces BLAS calls in standalone code.

`getBLASIntTypeName` is an abstract method. You must implement it in the definition of your callback class that derives from `coder.BLASCallback`.The generated code uses the integer data type name to specify types of variables in the generated code that produces BLAS calls.

Output Arguments

`intTypeName` — CBLAS integer data type name
character vector

Character vector that specifies the name of the integer data type that the CBLAS interface to a specific BLAS library uses.

Attributes

Abstract	true
Static	true

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

Return CBLAS Integer Data Type Name

This example shows how to write a `getBLASIntTypeName` method to return the name of the CBLAS integer data type.

In a class that derives from `coder.BLASCallback`, write a method `getBLASIntTypeName` that returns the name of the CBLAS integer data type as a character vector. This example is an implementation of the callback class `mklcallback` for integration with the Intel MKL BLAS library on a Windows platform. In this class, `getBLASIntTypeName` returns `'MKL_INT'`, which is the CBLAS integer data type for the Intel MKL BLAS library.

```
classdef mklcallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, ~)
            libPath = fullfile(pwd,'mkl','WIN','lib','intel64');
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libs = {'mkl_intel_ilp64.lib' 'mkl_intel_thread.lib' 'mkl_core.lib'};
            buildInfo.addLinkObjects(libs, libPath, libPriority, libPreCompiled, libLi
            buildInfo.addLinkObjects('libiomp5md.lib',fullfile(matlabroot,'bin','win64
                libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(fullfile(pwd,'mkl','WIN','include'));
            buildInfo.addDefines('-DMKL_ILP64');
        end
        function headerName = getHeaderFilename()
            headerName = 'mkl_cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
```

```
            intTypeName = 'MKL_INT';
        end
    end
end
```

If you are using a different BLAS library, replace `'MKL_INT'` with the name of your CBLAS integer data type.

See Also

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

http://www.netlib.org/blas/
https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines

Introduced in R2018b

coder.BLASCallback.getBLASDoubleComplexTypeName

Class: coder.BLASCallback
Package: coder

Return name of double-precision complex data type used by CBLAS interface

Syntax

```
doubleComplexTypeName =
coder.BLASCallback.getBLASDoubleComplexTypeName()
```

Description

`doubleComplexTypeName = coder.BLASCallback.getBLASDoubleComplexTypeName()` returns the name of the double-precision complex data type that is used by the CBLAS interface to a specific BLAS library.

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. At code generation time, if you specify a BLAS callback class, for certain vector and matrix function calls, the code generator produces BLAS calls in standalone code.

By default, the callback class that you define inherits the `getBLASDoubleComplexTypeName` method from `coder.BLASCallback`. If your BLAS library takes a type other than `double*` and `void*` for double-precision complex array arguments, you must override the inherited `getBLASDoubleComplexTypeName` method with your own implementation in your callback class definition.

The generated code uses the double-precision complex data type name to specify types of variables in the generated code that produces BLAS calls.

Output Arguments

`doubleComplexTypeName` — CBLAS double-precision complex data type name
character vector

Character vector that specifies the name of the double-precision complex data type that the CBLAS interface to a specific BLAS library uses.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

Return Double Complex Type Name

If your BLAS library takes a type other than `double*` and `void*` for double-precision complex array arguments, you must include this `Static` method in your callback class definition.

```
function doubleComplexTypeName = getBLASDoubleComplexTypeName()
doubleComplexTypeName = 'my_double_complex_type';
end
```

Replace `my_double_complex_type` with the type that your BLAS library takes for double-precision complex array arguments.

See Also

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

http://www.netlib.org/blas/

Introduced in R2018b

coder.BLASCallback.getBLASSingleComplexTypeName

Class: coder.BLASCallback
Package: coder

Return name of single-precision complex data type used by CBLAS interface

Syntax

```
singleComplexTypeName =
coder.BLASCallback.getBLASSingleComplexTypeName()
```

Description

`singleComplexTypeName = coder.BLASCallback.getBLASSingleComplexTypeName()` returns the name of the single-precision complex data type that is used by the CBLAS interface to a specific BLAS library.

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. At code generation time, if you specify a BLAS callback class, for certain vector and matrix function calls, the code generator produces BLAS calls in standalone code.

By default, the callback class that you define inherits the `getBLASSingleComplexTypeName` method from `coder.BLASCallback`. If your BLAS library takes a type other than `float*` and `void*` for single-precision complex array arguments, you must override the inherited `getBLASSingleComplexTypeName` method with your own implementation in your callback class definition.

The generated code uses the single-precision complex data type name to specify types of variables in the generated code that produces BLAS calls.

Output Arguments

`singleComplexTypeName` — CBLAS single-precision complex data type name
character vector

Character vector that specifies the name of the single-precision complex data type that the CBLAS interface to a specific BLAS library uses.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

Return Single Complex Type Name

If your BLAS library takes a type other than `float*` and `void*` for single-precision complex array arguments, you must include this `Static` method in your callback class definition.

```
function singleComplexTypeName = getBLASSingleComplexTypeName()
doubleComplexTypeName = 'my_single_complex_type';
end
```

Replace `my_single_complex_type` with the type that your BLAS library takes for single-precision complex array arguments.

See Also

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

http://www.netlib.org/blas/

Introduced in R2018b

coder.BLASCallback.useEnumNameRatherThanTypedef

Class: coder.BLASCallback
Package: coder

Specify whether types for enumerations in a BLAS library include the `enum` keyword

Syntax

```
p = coder.BLASCallback.useEnumNameRatherThanTypedef()
```

Description

`p = coder.BLASCallback.useEnumNameRatherThanTypedef()` returns `true` if types for enumerations in your BLAS library include the `enum` keyword. Otherwise it returns `false`.

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. At code generation time, if you specify a BLAS callback class, for certain vector and matrix function calls, the code generator produces BLAS calls in standalone code.

By default, the callback class that you define inherits the `useEnumNameRatherThanTypedef` method from `coder.BLASCallback` and returns `false`. If types for enumerations in your BLAS library include the `enum` keyword, you must override the inherited `useEnumNameRatherThanTypedef` method with your own implementation in your callback class definition. In such cases, the `useEnumNameRatherThanTypedef` method must return `true`.

The generated code uses the output of `useEnumNameRatherThanTypedef` to specify types of variables in the generated code that produces BLAS calls.

Output Arguments

p — Specify whether types for enumerations in BLAS library include the `enum` keyword
false | true

Logical variable that is `true` if types for enumerations in the BLAS library include the `enum` keyword, otherwise it is `false`.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes (MATLAB).

Examples

Specify Whether Types for Enumerations Include the `enum` Keyword

If types for enumerations in your BLAS library use the `enum` keyword, you must include this `Static` method in your callback class definition.

```
function p = useEnumNameRatherThanTypedef()
p = true;
end
```

See Also

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

http://www.netlib.org/blas/

Introduced in R2018b

coder.fftw.StandaloneFFTW3Interface.getNumThreads

Class: coder.fftw.StandaloneFFTW3Interface
Package: coder.fftw

Return number of threads to use for FFTW library calls

Syntax

```
coder.fftw.StandaloneFFTW3Interface.getNumThreads()
```

Description

`coder.fftw.StandaloneFFTW3Interface.getNumThreads()` returns the number of threads to use for calls to a specific FFTW library.

An FFT library callback class that derives from a `coder.fftw.StandaloneFFTW3Interface` class specifies the FFTW library to use.

Examples

Return Number of Threads to Use for FFTW Library Calls

In a class that derives from `coder.fft.StandaloneFFTW3Interface`, implement a method `getNumThreads` that returns the number of threads for the FFTW library to use.

Use the `getNumThreads` method in this example `coder.fftw.StandaloneFFTW3Interface` class as a template.

```
% copyright 2017 The MathWorks, Inc.

classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
```

```
            th = int32(coder.const(1));
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '/usr/lib/fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
             libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);
        end
    end
end
```

In your `getNumThreads` method, set `th` to the number of threads that you want to use. For example, this code specifies three threads:

```
th = int32(coder.const(3))
```

See Also

Topics

“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”
“Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.fftw.org

Introduced in R2017b

coder.fftw.StandaloneFFTW3Interface.getPlanMethod

Class: coder.fftw.StandaloneFFTW3Interface
Package: coder.fftw

Return FFTW planning method

Syntax

```
coder.fftw.StandaloneFFTW3Interface.getPlanMethod()
```

Description

`coder.fftw.StandaloneFFTW3Interface.getPlanMethod()` returns the FFTW planning method for FFTW library calls in generated standalone code.

When you define an FFTW library callback class that derives from a `coder.fftw.StandaloneFFTW3Interface` class, you do not have to implement a `getPlanMethod` method. By default, the planning method is `FFTW_ESTIMATE`. To use a different method, implement the `getPlanMethod` method. Specify one of the planning methods described in the planning section of the FFTW website.

Examples

Specify FFTW Planning Method

Specify the `FFTW_MEASURE` planning method in a `getPlanMethod` method in an FFTW library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.

```
% copyright 2017 The MathWorks, Inc.

classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
```

```
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function me = getPlanMethod
            coder.inline('always');
            me = coder.const(coder.opaque('int', 'FFTW_MEASURE'));
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '/usr/lib/fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
             libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);
        end
    end
end
```

See Also

Topics

“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”
“Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.fftw.org

Introduced in R2017b

coder.fftw.StandaloneFFTW3Interface.lock

Class: coder.fftw.StandaloneFFTW3Interface
Package: coder.fftw

Lock access to FFTW planning

Syntax

```
coder.fftw.StandaloneFFTW3Interface.lock()
```

Description

`coder.fftw.StandaloneFFTW3Interface.lock()` locks access to the planning process for FFTW library calls in generated standalone code.

When multiple threads call an FFTW library, implement this method in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.

Examples

Synchronize Multithreaded Access to FFTW Planning in Generated Code

In a class that derives from `coder.fft.StandaloneFFTW3Interface`, implement `lock` and `unlock` methods that call C code to manage a lock.

Write C functions that initialize, set, and unset a lock. To manage the lock, this example uses the OpenMP library. For a different library, modify the code accordingly.

- Create a file `mylock.c` that contains this C code:

```
#include "mylock.h"
#include "omp.h"

static omp_nest_lock_t lockVar;
```

```
void mylock_initialize(void)
{
    omp_init_nest_lock(&lockVar);
}

void mylock(void)
{
    omp_set_nest_lock(&lockVar);
}

void myunlock(void)
{
    omp_unset_nest_lock(&lockVar);
}
```

- Create a header file `mylock.h` that contains:

```
#ifndef MYLOCK_H
#define MYLOCK_H

 void mylock_initialize(void);
 void mylock(void);
 void myunlock(void);

#endif
```

Write an FFT callback class `myfftcb` that:

- Specifies the FFTW library.
- Implements `lock` and `unlock` methods that call the supporting C code to control access to the FFTW planning.

Use this class as a template. Replace `fftwLocation` with the location of your FFTW library installation.

```
classdef myfftcb < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function lock()
```

```
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('mylock');
        end

        function unlock()
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('myunlock');
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '\usr\lib\fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
            libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);

        end
    end
end
```

Set the code generation configuration parameters.

- For code generation with the MATLAB Coder `codegen` command, set:
  - `CustomFFTCallback` to `'myfftcb'`.
  - `CustomSource` to `'mylock.c'`.
  - `CustomInitializer` to `'mylock_initialize();'`.
- For code generation with the MATLAB Coder app, set:

  - **Custom FFT library callback** to `myfftcb`.
  - **Additional source files** to `mylock.c`.
  - **Initialize function** to `mylock_initialize();`.
- For code generation from a MATLAB Function block by using Simulink Coder, set these parameters:
  - **Custom FFT library callback** to `myfftcb`.
  - In **Code Generation > Custom Code**, under **Additional build information**, set **Source files** to `mylock.c`.
  - In **Code Generation > Custom Code**, under **Insert custom C code in generated**, set **Initialize function** to `mylock_initialize();`.

Generate code.

See Also

Topics

“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”
“Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code”
“Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)
“Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.fftw.org

Introduced in R2017b

coder.fftw.StandaloneFFTW3Interface.unlock

Class: coder.fftw.StandaloneFFTW3Interface
Package: coder.fftw

Unlock access to FFTW planning

Syntax

```
coder.fftw.StandaloneFFTW3Interface.unlock()
```

Description

`coder.fftw.StandaloneFFTW3Interface.unlock()` unlocks access to planning for FFTW library calls in generated standalone code.

When multiple threads call an FFTW library, implement this method in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.

Examples

Synchronize Multithreaded Access to FFTW Planning in Generated Code

In a class that derives from `coder.fft.StandaloneFFTW3Interface`, implement `lock` and `unlock` methods that call C code to manage a lock.

Write C functions that initialize, set, and unset a lock. To manage the lock, this example uses the OpenMP library. For a different library, modify the code accordingly.

- Create a file `mylock.c` that contains this C code:

```
#include "mylock.h"
#include "omp.h"

static omp_nest_lock_t lockVar;
```

```
void mylock_initialize(void)
{
    omp_init_nest_lock(&lockVar);
}

void mylock(void)
{
    omp_set_nest_lock(&lockVar);
}

void myunlock(void)
{
    omp_unset_nest_lock(&lockVar);
}
```

- Create a header file `mylock.h` that contains:

```
#ifndef MYLOCK_H
#define MYLOCK_H

 void mylock_initialize(void);
 void mylock(void);
 void myunlock(void);

#endif
```

Write an FFT callback class `myfftcb` that:

- Specifies the FFTW library.
- Implements `lock` and `unlock` methods that call the supporting C code to control access to the FFTW planning.

Use this class as a template. Replace `fftwLocation` with the location of your FFTW library installation.

```
classdef myfftcb < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function lock()
```

```
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('mylock');
        end

        function unlock()
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('myunlock');
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '\usr\lib\fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
            libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);

        end
    end
end
```

Set the code generation configuration parameters.

- For code generation with the MATLAB Coder `codegen` command, set:
  - `CustomFFTCallback` to `'myfftcb'`.
  - `CustomSource` to `'mylock.c'`.
  - `CustomInitializer` to `'mylock_initialize();'`.
- For code generation with the MATLAB Coder app, set:

- **Custom FFT library callback** to `myfftcb`.
- **Additional source files** to `mylock.c`.
- **Initialize function** to `mylock_initialize();`.

- For code generation from a MATLAB Function block by using Simulink Coder, set these parameters:
  - **Custom FFT library callback** to `myfftcb`.
  - In **Code Generation > Custom Code**, under **Additional build information**, set **Source files** to `mylock.c`.
  - In **Code Generation > Custom Code**, under **Insert custom C code in generated**, set **Initialize function** to `mylock_initialize();`.

Generate code.

See Also

Topics

“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”
“Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code”
“Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)
“Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.fftw.org

Introduced in R2017b

coder.fftw.StandaloneFFTW3Interface.updateBuildInfo

Class: coder.fftw.StandaloneFFTW3Interface
Package: coder.fftw

Update the build information for linking to a specific FFTW library

Syntax

```
coder.fftw.StandaloneFFTW3Interface.updateBuildInfo(buildInfo, ctx)
```

Description

`coder.fftw.StandaloneFFTW3Interface.updateBuildInfo(buildInfo, ctx)` updates the build information to link to a specific FFTW library.

An FFT library callback class that derives from a `coder.fftw.StandaloneFFTW3Interface` class specifies the FFTW library.

Input Arguments

`buildInfo` — Build information object

build information object

After code generation, the build information object contains standard project, build option, and dependency information. In the `updateBuildInfo` method, to add the information for linking to a specific FFTW library, use build information methods.

`ctx` — `coder.BuildConfig` object

`coder.BuildConfig` object

Implement the `coder.BuildConfig` `getStdLibInfo` method to get the platform-specific file extension to use at link time.

Examples

Link to a Specific FFTW Library

In a class that derives from `coder.fft.StandaloneFFTW3Interface`, implement a method `updateBuildInfo` that updates the build information to link to a specific FFTW library.

Use the `updateBuildInfo` method in this example `coder.fftw.StandaloneFFTW3Interface` class as a template.

```
% copyright 2017 The MathWorks, Inc.

classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '/usr/lib/fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
             libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);
        end
    end
end
```

In your `updateBuildInfo` method, set:

- `fftwLocation` to the full path for your installation of the library.
- `includePath` to the full path of the folder that contains the FFTW library header file.

- `libPath` to the full path of the folder that contains the library files.

See Also

Topics

“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”
“Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.fftw.org

Introduced in R2017b

packNGo

Package generated code in zip file for relocation

Syntax

```
packNGo(buildInfo,{Name,Value})
```

Description

`packNGo(buildInfo,{Name,Value})` packages the code files in a compressed zip file so that you can relocate, unpack, and rebuild them in another development environment. The list of name-value pairs is optional.

The types of code files in the zip file include:

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- MAT-file that contains the build information object (`.mat` file)
- Nonbuild-related files (for example, `.dll` files and `.txt` informational files) required for a final executable
- Build-generated binary files (for example, executable `.exe` file or dynamic link library `.dll`).

  The code generator includes the build-generated binary files (if present) in the zip file. The **ignoreFileMissing** property does not apply to build-generated binary files.

Use this function to relocate files. You can then recompile the files for a specific target environment or rebuild them in a development environment in which MATLAB is not installed. By default, the function packages the files as a flat folder structure in a zip file within the code generation folder. You can customize the output by specifying name-value pairs. After relocating the zip file, use a standard zip utility to unpack the compressed file.

The `packNGo` function can potentially modify the build information passed in the first `packNGo` argument. As part of code packaging, `packNGo` can find additional files from

source and include paths recorded in the build information. When these files are found, `packNGo` adds them to the build information.

Examples

Run packNGo from Command Window

After the build process is complete, you can run packNGo from the Command Window. Use `packNGo` for zip file packaging of generated code in the file `portzingbit.zip`. Maintain the relative file hierarchy.

1. Change folders to the code generation folder. For example, using MATLAB Coder, `codegen/dll/zingbit`, or for Simulink code generation, `zingbit_grt_rtw`.
2. Load the `buildInfo` object that describes the build.
3. Run `packNGo` with property settings for `packType` and `fileName`.

```
cd codegen/dll/zingbit;
load buildInfo.mat
packNGo(buildInfo,{'packType', 'hierarchical', ...
   'fileName','portzingbit'});
```

Configure packNGo in the Simulink Editor

If you configure zip file packaging from the code generation pane, the code generator uses `packNGo` to output a zip file during the build process.

1. Select **Code Generation** > **Package code and artifacts**. Optionally, provide a **Zip file name**. To apply the changes, click **OK**.
2. Build the model. At the end of the build process, the code generator outputs the zip file. The folder structure in the zip file is hierarchical.

Configure packNGo for Simulink from the Command Line

If you configure zip file packaging with `set_param`, the code generator uses `packNGo` to output a zip file during the build process.

Use `set_param` to configure zip file packaging for model `zingbit` in the file `zingbit.zip`.

```
set_param('zingbit','PostCodeGenCommand', ...
   'packNGo(buildInfo);');
```

Input Arguments

`buildInfo` — Object that provides build information
`buildInfo` object

During the build process, the code generator places `buildInfo.mat` in the code generation folder. This MAT-file contains the `buildInfo` object. The object provides information that `packNGo` uses to produce the zip file.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `{'packType','flat','nestedZipFiles',true}`

`packType` — Determines whether the primary zip file contains secondary zip files or folders
`'flat'` (default) | `'hierarchical'`

If `'flat'`, package the generated code files in a zip file as a single, flat folder.

If `'hierarchical'`, package the generated code files hierarchically in a primary zip file.

Example: `{'packType','flat'}`

`nestedZipFiles` — Determines whether the paths for files in the secondary zip files are relative to the root folder of the primary zip file
`true` (default) | false

If `true`, create a primary zip file that contains three secondary zip files:

- `mlrFiles.zip` — Files in your *`matlabroot`* folder tree

- `sDirFiles.zip` — Files in and under your code generation folder
- `otherFiles.zip` — Required files not in the *`matlabroot`* or `start` folder trees

If `false`, create a primary zip file that contains folders, for example, your code generation folder and *`matlabroot`*.

Example: `{'nestedZipFiles',true}`

`fileName` — Specifies a file name for the primary zip file
`'`*`modelOrFunctionName`*`.zip'` (default) | `'`*`myName`*`'`

If you do not specify the `'fileName'`-value pair, the function packages the files in a zip file named *`modelOrFunctionName`*`.zip` and places the zip file in the code generation folder.

If you specify `'fileName'` with the value, `'`*`myName`*`'`, the function creates *`myName`*`.zip` in the code generation folder.

To specify another location for the primary zip file, provide the absolute path to the location, *`fullPath/myName`*`.zip`

Example: `{'fileName','/home/user/myModel.zip'}`

`minimalHeaders` — Selects whether to include only the minimal header files
`true` (default) | `false`

If `true`, include only the minimal header files required to build the code in the zip file.

If `false`, include header files found on the include path in the zip file.

Example: `{'minimalHeaders',true}`

`includeReport` — Selects whether to include the `html` folder for your code generation report
`false` (default) | `true`

If `false`, do not include the `html` folder in the zip file.

If `true`, include the `html` folder in the zip file.

Example: `{'includeReport',false}`

`ignoreParseError` — Instruct packNGo not to terminate on parse errors
false (default) | true

If `false`, terminate on parse errors.

If `true`, do not terminate on parse errors.

Example: `{'ignoreParseError',false}`

`ignoreFileMissing` — Instruct packNGo not to terminate if files are missing
false (default) | true

If `false`, terminate on missing file errors.

If `true`, do not terminate on missing files errors.

Example: `{'ignoreFileMissing',false}`

Limitations

- The function operates on source files only, such as `*.c`, `*.cpp`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if those files are not used.
- For MATLAB Coder, the function does not package example main source and header files that you generate with the default configuration settings. To package the example main files, configure code generation to generate and compile the example main function, generate your code, and then package the build files.
- `packNGo` does not package the code generated for MEX targets.

See Also

Topics

“Build Process Customization”
“Package Code for Other Development Environments”

Introduced in R2006b

getLineColumn

Package: coder

Find locations of beginning and end of MATLAB code involved in code generation

Syntax

```
[startLoc,endLoc] = getLineColumn(obj)
[startLoc,endLoc] = getLineColumn(obj_message)
```

Description

`[startLoc,endLoc] = getLineColumn(obj)` returns the line and column indices of the first and last character of the MATLAB code described by `obj` in the text of the file containing the code. The code described by `obj` is a MATLAB function or method that is involved in code generation.

`[startLoc,endLoc] = getLineColumn(obj_message)` returns the line and column indices of the first and last character of the MATLAB code that caused the code generation message described by `obj_message`.

Examples

Locate a MATLAB Function in File

Create a report information object for a code generation process. You then locate a MATLAB function involved in code generation in the file containing that function.

Define the MATLAB function `foo`:

```
function [b,c] = foo(a)
b = svd(a,0);
c = bar(a);
end
```

```
function c = bar(a)
c = inv(a);
end
```

Generate a static C library for `foo`. Specify the input as a string scalar. Export the code generation report information to the variable `info` in your base MATLAB workspace.

```
codegen -config:lib foo -args {"A string scalar"} -reportinfo info
```

Code generation fails because a string scalar is not a valid input for the MATLAB functions `svd` and `inv`. The code generator creates a report information object `info` in the base MATLAB workspace.

The property `info.Functions` is a two-dimensional array. `info.Functions(1)` contains the description of the MATLAB function `foo`. `info.Functions(2)` contains the description of the MATLAB function `bar`.

To manually inspect the function `bar`, first display the text of the file containing `bar`.

```
info.Functions(2).File.Text
```

```
    'function [b,c] = foo(a)
     b = svd(a,0);
     c = bar(a);
     end

     function c = bar(a)
     c = inv(a);
     end
     '
```

Use `getLineColumn` to locate the beginning and end of the function `bar` in this text. The output `startLoc` contains the line and column indices of the first character of `bar`. The output `endLoc` contains the line and column indices of the last character of `bar`.

```
[startLoc,endLoc]=getLineColumn(info.Functions(2))
```

```
startLoc =

  struct with fields:

      Line: 6
    Column: 1
```

```
endLoc =

  struct with fields:

      Line: 8
    Column: 3
```

Locate MATLAB Code That Causes a Code Generation Error Message

Create a report information object for a code generation process that fails. You then locate the part of MATLAB code that caused an error message.

Define the MATLAB function `foo`:

```
function b = foo(a)
b = svd(a,0);
end
```

Generate a static C library for `foo`. Specify the input as a string scalar. Export the code generation report information to the variable `info` in your base MATLAB workspace.

```
codegen -config:lib foo -args {"A string scalar"} -reportinfo info
```

Code generation fails because a string scalar is not a valid input for the MATLAB function `svd`. The code generator creates a report information object `info` in the base MATLAB workspace.

The property `info.Messages` is a two-dimensional array containing descriptions of two code generation messages. Inspect the description of the first message.

```
info.Messages(1)
```

```
  Message with properties:

    Identifier: 'Coder:toolbox:unsupportedClass'
          Type: 'Error'
          Text: 'Function 'svd' is not defined for values of class 'string'.'
          File: [1×1 coder.CodeFile]
    StartIndex: 26
      EndIndex: 33
```

To manually inspect the segment of MATLAB code that caused this error message, first display the text of the file associated with this error message.

```
info.Messages(1).File.Text
```

```
'function b = foo(a)
 b = svd(a,0);
 end
 '
```

Use `getLineColumn` to locate the beginning and end of the part of the code that caused the error message. The output `startLoc` contains the line and column indices of the first character of the code segment. The output `endLoc` contains the line and column indices of the last character of the code segment.

```
[startLoc,endLoc] = getLineColumn(info.messages(1))
```

```
startLoc =

  struct with fields:

      Line: 2
    Column: 5


endLoc =

  struct with fields:

      Line: 2
    Column: 12
```

These locations correspond to the beginning and the end of the function call `'svd(a,0)'` in the text of `foo.m`.

Input Arguments

`obj` — Object describing MATLAB function or method involved in code generation

handle object

Object describing a MATLAB function or a method in a MATLAB class that is involved in code generation, specified as one of the following:

- A `coder.Function` object for the description of a function. See coder.Function Properties.
- A `coder.Method` object for the description of a method. See coder.Method Properties.

`obj_message` — Object describing code generation error message
handle object

A `coder.Message` object describing an error, warning, or informational message produced during code generation from MATLAB code. See coder.Message Properties.

Output Arguments

`startLoc` — Line and column indices of the first character of MATLAB code
structure array

Structure array with two fields: `Line` and `Column`.

- `startLoc.Line` is the line index of the first character of the MATLAB code in the text of the file containing the code.
- `startLoc.Column` is the column index of the first character of the MATLAB code in the text of the file containing the code.

`endLoc` — Line and column indices of the last character of MATLAB code
structure array

Structure array with two fields: `Line` and `Column`.

- `endLoc.Line` is the line index of the last character of the MATLAB code in the text of the file containing the code.
- `endLoc.Column` is the column index of the last character of the MATLAB code in the text of the file containing the code.

See Also

coder.Function Properties | coder.Message Properties | coder.Method Properties

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

3

Class Reference

coder.ArrayType class

Package: coder
Superclasses:

Represent set of MATLAB arrays

Description

Specifies the set of arrays that the generated code accepts. Use only with the `codegen -args` option. Do not pass as an input to a generated MEX function.

Construction

`coder.ArrayType` is an abstract class. You cannot create instances of it directly. You can create `coder.EnumType`, `coder.FiType`, `coder.PrimitiveType`, and `coder.StructType` objects that derive from this class.

Properties

`ClassName`

Class of values in this set

`SizeVector`

The upper-bound size of arrays in this set.

`VariableDims`

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

`codegen` | `coder.CellType` | `coder.ClassType` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof`

Introduced in R2011a

coder.BuildConfig class

Package: coder

Build context during code generation

Description

The code generator creates an object of this class to facilitate access to the build context. The build context encapsulates the settings used by the code generator including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

Use `coder.BuildConfig` methods in the methods that you write for the `coder.ExternalDependency` class.

Construction

The code generator creates objects of this class.

Methods

getHardwareImplementation	Get handle of copy of hardware implementation object
getStdLibInfo	Get standard library information
getTargetLang	Get target code generation language
getToolchainInfo	Returns handle of copy of toolchain information object
isCodeGenTarget	Determine if build configuration represents specified target
isMatlabHostTarget	Determine if hardware implementation object target is MATLAB host computer

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods

This example shows how to use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods. In this example, you use:

- `coder.BuildConfig.isMatlabHostTarget` to verify that the code generation target is the MATLAB host. If the host is not MATLAB report an error.
- `coder.BuildConfig.getStdLibInfo` to get the link-time and run-time library file extensions. Use this information to update the build information.

Write a class definition file for an external library that contains the function `adder`.

```
%==========================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%==========================================================
```

```
classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(buildContext)
            if  buildContext.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, buildContext)
            % Get file extensions for the current platform
            [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

            % Add file paths
            hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
            buildInfo.addIncludePaths(hdrFilePath);

            % Link files
            linkFiles = strcat('adder', linkLibExt);
            linkPath = hdrFilePath;
            linkPriority = '';
            linkPrecompiled = true;
            linkLinkOnly = true;
            group = '';
            buildInfo.addLinkObjects(linkFiles, linkPath, ...
                linkPriority, linkPrecompiled, linkLinkOnly, group);

            % Non-build files for packaging
            nbFiles = 'adder';
            nbFiles = strcat(nbFiles, execLibExt);
            buildInfo.addNonBuildFiles(nbFiles,'','');
        end

        %API for library function 'adder'
        function c = adder(a, b)
```

```
            if coder.target('MATLAB')
                % running in MATLAB, use built-in addition
                c = a + b;
            else
                % Add the required include statements to the generated function code
                coder.cinclude('adder.h');
                coder.cinclude('adder_initialize.h');
                coder.cinclude('adder_terminate.h');
                c = 0;

                % Because MATLAB Coder generated adder, use the
                % housekeeping functions before and after calling
                % adder with coder.ceval.

                coder.ceval('adder_initialize');
                c = coder.ceval('adder', a, b);
                coder.ceval('adder_terminate');
            end
        end
    end
end
```

See Also

`coder.ExternalDependency` | `coder.HardwareImplementation` | `coder.make.ToolchainInfo` | `coder.target`

Topics

“Develop Interface for External C/C++ Code”
“Build Process Customization”

Introduced in R2013b

coder.CellType class

Package: coder
Superclasses:

Represent set of MATLAB cell arrays

Description

Specifies the set of cell arrays that the generated code accepts. Use only with the `codegen -args` option. Do not pass as an input to a generated MEX function.

Construction

`t = coder.typeof(cells)` creates a `coder.CellType` object for a cell array that has the same cells and cell types as `cells`. The cells in `cells` are type objects or example values.

`t = coder.typeof(cells, sz, variable_dims)` creates a `coder.CellType` object that has upper bounds specified by `sz` and variable dimensions specified by `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds do not change. If you do not specify the `variable_dims` input parameter, except for the unbounded dimensions, the dimensions of the type are fixed. A scalar `variable_dims` applies to the bounded dimensions that are not `1` or `0`.

When `cells` specifies a cell array whose elements have different classes, you cannot use `coder.typeof` to create a `coder.CellType` object for a variable-size cell array.

`t = coder.newtype(cells)` creates a `coder.CellType` object for a cell array that has the cells and cell types specified by `cells`. The cells in `cells` must be type objects.

`t = coder.newtype(cell_array, sz, variable_dims)` creates a `coder.CellType` that has upper bounds specified by `sz` and variable dimensions specified by `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds do not change. If you do not specify the `variable_dims` input parameter, except

for the unbounded dimensions, the dimensions of the type are fixed. A scalar `variable_dims` applies to the bounded dimensions that are not `1` or `0`.

When `cells` specifies a cell array whose elements have different classes, you cannot use `coder.newtype` to create a `coder.CellType` object for a variable-size cell array.

Input Arguments

`cells` — Specification of cell types

cell array

Cell array that specifies the cells and cell types for the output `coder.CellType` object. For `coder.typeof`, `cells` can contain type objects or example values. For `coder.newtype`, `cells` must contain type objects.

`sz` — Size of cell array

row vector of integer values

Specifies the upper bound for each dimension of the cell array type object. For `coder.newtype`, `sz` cannot change the number of cells for a heterogeneous cell array.

For `coder.newtype`, the default is `[1 1]`.

`variable_dims` — Dimensions that are variable size

row vector of logical values

Specifies whether each dimension is variable size (true) or fixed size (false).

For `coder.newtype`, the default is `true` for dimensions for which `sz` specifies an upper bound of `inf` and `false` for all other dimensions.

When `cells` specifies a cell array whose elements have different classes, you cannot create a `coder.CellType` object for a variable-size cell array.

Properties

`Alignment` — Run-time memory alignment

`-1` | power of 2 that is less than or equal to 128

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide

the ability to align data objects passed into a replacement function to a specified boundary. You can take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary, so it is not matched by CRL functions that require alignment.

`Cells` — Types of cells
cell array

A cell array that specifies the `coder.Type` of each cell.

`ClassName` — Name of class
character vector or string scalar

Class of values in this set.

`Extern` — External definition
logical scalar

Specifies whether the cell array type is externally defined.

`HeaderFile` — Name of header file
nonempty character vector or string scalar

If the cell array type is externally defined, the name of the header file that contains the external definition of the type, for example, `'mytype.h'`. If you use the `codegen` command to specify the path to the file, use the `-I` option. If you use the MATLAB Coder app to specify the path to the file, use the **Additional include directories** setting in the **Custom Code** tab in the project settings dialog box.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. If you specify the `HeaderFile` option, the code generator includes the custom header file where it is required.

`SizeVector` — Size of cell array
row vector of integer values

The upper bounds of dimensions of the cell array.

`TypeName` — Name of generated structure type
character vector

The name to use in the generated code for the structure type that represents this cell array type. `TypeName` applies only to heterogeneous cell arrays types.

`VariableDims` — Dimensions that are variable size
row vector of logical values

A vector that specifies whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Methods

isHeterogeneous	Determine whether cell array type represents a heterogeneous cell array
isHomogeneous	Determine whether cell array type represents a homogeneous cell array
makeHeterogeneous	Make a heterogeneous copy of a cell array type
makeHomogeneous	Create a homogeneous copy of a cell array type

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a Type for a Cell Array Whose Elements Have the Same Class

Create a type for a cell array whose first element has class char and whose second element has class double.

```
t = coder.typeof({1 2 3})
```

```
t =

coder.CellType
   1x3 homogeneous cell
      base: 1x1 double
```

The type is homogeneous.

Create a Heterogeneous Type for a Cell Array Whose Elements Have the Same Class

To create a heterogeneous type when the elements of the example cell array type have the same class, use the `makeHeterogeneous` method.

```
t = makeHeterogeneous(coder.typeof({1 2 3}))
```

```
t =

coder.CellType
   1×3 locked heterogeneous cell
      f1: 1×1 double
      f2: 1×1 double
      f3: 1×1 double
```

The cell array type is heterogeneous. It is represented as a structure in the generated code.

Create a Cell Array Type for a Cell Array Whose Elements Have Different Classes

Define variables that are example cell values.

```
a = 'a';
b = 1;
```

Pass the example cell values to `coder.typeof`.

```
t = coder.typeof({a, b})
```

```
t =

coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 char
      f1: 1x1 double
```

Create a Type for a Variable-Size Homogeneous Cell Array from an Example Cell Array Whose Elements Have Different Classes

Create a type for a cell array that contains two character vectors that have different sizes.

```
t = coder.typeof({'aa', 'bbb'})
```

```
t =

coder.CellType
   1x2 heterogeneous cell
      f0: 1x2 char
      f1: 1x3 char
```

The cell array type is heterogeneous.

Create a type using the same cell array input. This time, specify that the cell array type has variable-size dimensions.

```
t = coder.typeof({'aa','bbb'},[1,10],[0,1])
```

```
t =

coder.CellType
   1×:10 locked homogeneous cell
      base: 1×:3 char
```

The cell array type is homogeneous. `coder.typeof` determined that the base type `1x:3 char` can represent `'aa'`, and `'bbb'`.

Create a New Cell Array Type from a Cell Array of Types

Create a type for a scalar `int8`.

```
ta = coder.newtype('int8',[1 1]);
```

Create a type for a :1x:2 double row vector.

```
tb = coder.newtype('double',[1 2],[1 1]);
```

Create a cell array type whose cells have the types specified by `ta` and `ta`.

```
t = coder.newtype('cell',{ta,tb})
```

```
t =

coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 int8
      f1: :1x:2 double
```

Create a `coder.CellType` That Uses an Externally Defined Type

Create a cell type for a heterogeneous cell array.

```
ca = coder.typeof(double(0));
cb = coder.typeof(single(0));
t = coder.typeof({ca cb})
```

```
coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 double
      f1: 1x1 single
```

Use `coder.cstructname` to specify the name for the type and that the type is defined in an external file.

```
t = coder.cstructname(t,'mytype','extern','HeaderFile','myheader.h')
```

```
t =

coder.CellType
   1×2 extern locked heterogeneous cell mytype(myheader.h) [-1]
      f1: 1×1 double
      f2: 1×1 single
```

Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size,

`coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

See Also

`codegen` | `coder.ArrayType` | `coder.ClassType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof`

Topics

“Code Generation for Cell Arrays”

Introduced in R2015b

coder.ClassType class

Package: coder
Superclasses:

Represent set of MATLAB classes

Description

Specifies the set of value class objects that the generated code can accept. Use only with the `codegen -args` option. Do not pass as an input to a generated MEX function.

Construction

`t = coder.typeof(value_class_object)` creates a `coder.ClassType` object for the object `value_class_object`.

`t = coder.newtype(value_class_name)` creates a `coder.ClassType` object for an object of the class `value_class_name`.

Input Arguments

`value_class_object`

Value class object from which to create the `coder.ClassType` object. `value_class_object` is an expression that evaluates to an object of a value class. For example:

```
v = myValueClass;
t = coder.typeof(v);

t = coder.typeof(myValueClass(2,3));
```

`value_class_name`

Name of a value class definition file on the MATLAB path. Specify as a character vector or string scalar. For example:

```
t = coder.newtype('myValueClass');
```

Properties

When you create a `coder.ClassType` object `t` from a value class object `v` by using `coder.typeof`, the properties of `t` are the same as the properties of `v` with the attribute `Constant` set to `false`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create Type Based on Example Object

Create a type based on an example object in the workspace.

Create a value class `myRectangle`.

```
classdef myRectangle
    properties
        length;
        width;
    end
    methods
        function obj = myRectangle(l,w)
            if nargin > 0
                obj.length = l;
                obj.width = w;
            end
        end
        function area = calcarea(obj)
            area = obj.length * obj.width;
        end
    end
end
```

Create a function that takes an object of `myRectangle` as an input.

```
function z = getarea(r)
%#codegen
z = calcarea(r);
end
```

Create an object of `myRectangle`.

```
v = myRectangle(1,2)
```

```
v =

  myRectangle with properties:

    length: 1
     width: 2
```

Create a `coder.ClassType` object based on `v`.

```
t = coder.typeof(v)
```

```
t =

coder.ClassType
   1×1 myRectangle
      length: 1×1 double
      width : 1×1 double
```

`coder.typeof` creates a `coder.ClassType` object that has the same properties names and types as `v` has.

Generate code for `getarea`. Specify the input type by passing the `coder.ClassType` object, `t`, to the `-args` option.

```
codegen getarea -args {t} -report
```

Create Type by Using `coder.newtype`

Create a `coder.ClassType` object for an object of the value class `mySquare` by using `coder.newtype`.

Create value class `mySquare` that has one property, `side`.

```
classdef mySquare
    properties
```

```
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end
```

Create a `coder.ClassType` type for `mySquare`.

```
t = coder.newtype('mySquare')
```

Specify the type of `side`.

```
t.Properties.side = coder.typeof(2)
```

Tips

- After you create a `coder.ClassType`, you can modify the types of the properties. For example:

  ```
  t = coder.typeof(myClass)
  t.Properties.prop1 = coder.typeof(int16(2));
  t.Properties.prop2 = coder.typeof([1 2 3]);
  ```

- After you create a `coder.ClassType`, you can add properties. For example:

  ```
  t = coder.typeof(myClass)
  t.Properties.newprop1 = coder.typeof(int8(2));
  t.Properties.newprop2 = coder.typeof([1 2 3]);
  ```

- When you generate code, the properties of the `coder.ClassType` object that you pass to `codegen` must be consistent with the properties in the class definition file. However, if the class definition file has properties that your code does not use, the `coder.ClassType` object does not have to include those properties. The code generator removes properties that you do not use.

See Also

codegen | coder | coder.ArrayType | coder.Constant | coder.EnumType | coder.EnumType | coder.FiType | coder.PrimitiveType | coder.Type | coder.cstructname | coder.newtype | coder.resize | coder.typeof

Introduced in R2017a

coder.Constant class

Package: coder
Superclasses:

Represent set containing one MATLAB value

Description

Use a `coder.Constant` object to define values that are constant during code generation. Use only with the `codegen -args` options. Do not pass as an input to a generated MEX function.

Construction

`const_type=coder.Constant(v)` creates a `coder.Constant` type from the value `v`.

`codegen -globals {'g', coder.Constant(v)}` creates a constant global variable `g` with the value `v`.

`const_type=coder.newtype('constant', v)` creates a `coder.Constant` type from the value `v`.

Input Arguments

`v`

Constant value used to construct the type.

Properties

`Value`

The actual value of the constant.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Generate MEX code for a MATLAB function with a constant input

This example shows how to generate MEX code for a MATLAB function that has a constant input. It shows how to use the `ConstantInputs` configuration parameter to control whether the MEX function signature includes constant inputs and whether the constant input values must match the compile-time values.

Write a function `myadd` that returns the sum of its inputs.

```
function c = myadd(a,b)
c = a + b;
end
```

Create a configuration object for MEX code generation.

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

Look at the value of the constant input checking configuration parameter.

```
mexcfg.ConstantInputs
```

```
ans =

    'CheckValues'
```

It has the default value.

Generate a MEX function `myadd_mex`. Specify that the first argument is a double scalar and the second argument is a constant with value `3`.

```
codegen myadd -config mexcfg -args {1, coder.Constant(3)}
```

Call `myadd_mex`. You must provide the input `3` for the second argument.

```
myadd_mex(1,3)
```

```
ans =

    4
```

Modify `ConstantInputs` so that the MEX function does not check that the input value matches the value specified at code generation time.

```
mexcfg.ConstantInputs = 'IgnoreValues';
```

Generate `myadd_mex`.

```
codegen myadd -config mexcfg -args {1, coder.Constant(3)}
```

Call `myadd_mex` with a constant input value other than `3`, for example `5`.

```
myadd_mex(1,5)
```

```
ans =

    4
```

The MEX function ignores the input value `5`. It uses the value `3`, which is the value that you specified for the constant argument `b` when you generated `myadd_mex`.

Modify `ConstantInputs` so that the MEX function signature does not include the constant input argument.

```
mexcfg.ConstantInputs = 'Remove';
```

Generate `myadd_mex`.

```
codegen myadd -config mexcfg -args {1, coder.Constant(3)}
```

Call `myadd_mex`. Provide the value `1` for `a`. Do not provide a value for the constant argument `b`.

```
myadd_mex(1)
```

```
ans =

    4
```

Generate C code for a function that has constant input

This example shows how to generate C code for a function specialized to the case where an input has a constant value.

Write a function `identity` that copies its input to its output.

```
function y = identity(u) %#codegen
y = u;
```

Create a code configuration object for C code generation.

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

Generate C code for `identity` with the constant input `42` and generate a report.

```
codegen identity -config cfg -args {coder.Constant(42)} -report
```

In the report, on the **C code** tab, click `identity.c`.

The function signature for `identity` is

```
double identity(void)
```

Generate MEX code for a function that uses constant global data

This example shows how to specify a constant value for a global variable at compile time.

Write a function `myfunction` that returns the value of the global constant `g`.

```
function  y = myfunction() %#codegen
global g;

y = g;

end
```

Create a configuration object for MEX code generation.

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

Define a cell array `globals` that declares that `g` is a constant global variable with value `5`.

```
globals = {'g', coder.Constant(5)};
```

Generate a MEX function for `myfunction` using the `-globals` option to specify the global data.

```
codegen -config cfg -globals globals myfunction
```

Run the generated MEX function.

```
myfunction_mex
```

```
ans =

     5
```

Limitations

- You cannot use `coder.Constant` on sparse matrices, or on structures, cell arrays, or classes that contain sparse matrices.

See Also

`codegen` | `coder.Type` | `coder.newtype`

Topics

“Specify Constant Inputs at the Command Line”
“Constant Input Checking in MEX Functions”
“Define Constant Global Data”

Introduced in R2011a

coder.EnumType class

Package: coder
Superclasses:

Represent set of MATLAB enumerations

Description

Specifies the set of MATLAB enumerations that the generated code should accept. Use only with the `codegen -args` options. Do not pass as an input to a generated MEX function.

Construction

`enum_type = coder.typeof(enum_value)` creates a `coder.EnumType` object representing a set of enumeration values of class (`enum_value`).

`enum_type = coder.typeof(enum_value, sz, variable_dims)` returns a modified copy of `coder.typeof(enum_value)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is [], the (upper bound) sizes of `v` do not change. If you do not specify `variable_dims`, the bounded dimensions of the type are fixed; the unbounded dimensions are variable size. When `variable_dims` is a scalar, it applies to bounded dimensions that are not `1` or `0` (which are fixed).

`enum_type = coder.newtype(enum_name,sz,variable_dims)` creates a `coder.EnumType` object that has variable size with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. If you do not specify `variable_dims`, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to bounded dimensions that are not `1` or `0` (which are fixed).

Input Arguments

`enum_value`

Enumeration value defined in a file on the MATLAB path.

`sz`

Size vector specifying each dimension of type object.

Default: [1 1] for `coder.newtype`

`variable_dims`

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

Default: false(size(sz)) | sz==Inf for `coder.newtype`

`enum_name`

Name of enumeration defined in a file on the MATLAB path.

Properties

`ClassName`

Class of values in the set.

`SizeVector`

The upper-bound size of arrays in the set.

`VariableDims`

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a `coder.EnumType` object using a value from an existing MATLAB enumeration.

1. Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

2. Create a `coder.EnumType` object from this enumeration.

```
t = coder.typeof(MyColors.red);
```

Create a `coder.EnumType` object using the name of an existing MATLAB enumeration.

1. Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

2. Create a `coder.EnumType` object from this enumeration.

```
t = coder.newtype('MyColors');
```

See Also

`codegen` | `coder.ArrayType` | `coder.ClassType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof`

Topics

“Enumerations”

Introduced in R2011a

coder.ExternalDependency class

Package: coder

Interface to external code

Description

`coder.ExternalDependency` is an abstract class for developing an interface between external code and MATLAB code intended for code generation. You can define classes that derive from `coder.ExternalDependency` to encapsulate the interface to external libraries, object files, and C/C++ source code. This encapsulation allows you to separate the details of the interface from your MATLAB code.

To define a class derived from `coder.ExternalDependency`, create a subclass. For example:

```
classdef myClass < coder.ExternalDependency
```

You must define all of the methods listed in “Methods” on page 3-31. These methods are static and are not compiled. The code generator invokes these methods in MATLAB after code generation is complete to configure the build for the generated code. The `RTW.BuildInfo` and `coder.BuildConfig` objects that describe the build information and build context are automatically created during the build process. The `updateBuildInfo` method provides access to these objects. For more information on build information customization, see “Build Process Customization”.

You also define methods that call the external code. These methods are compiled. For each external function that you want to call, write a method to define the programming interface to the function. In the method, use `coder.ceval` to call the external function.

Methods

getDescriptiveName	Return descriptive name for external dependency
isSupportedContext	Determine if build context supports external dependency
updateBuildInfo	Update build information

Examples

Encapsulate the Interface to an External C Dynamic Library

This example shows how to encapsulate the interface to an external C dynamic linked library using `coder.ExternalDependency`.

Write a function `adder` that returns the sum of its inputs.

```
function c = adder(a,b)
%#codegen
c = a + b;
end
```

Generate a library that contains `adder`.

```
codegen('adder','-args',{-2,5},'-config:dll','-report')
```

Write the class definition file `AdderAPI.m` to encapsulate the library interface.

```
%=================================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=================================================================

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end
```

```
function tf = isSupportedContext(buildContext)
    if  buildContext.isMatlabHostTarget()
        tf = true;
    else
        error('adder library not available for this target');
    end
end

function updateBuildInfo(buildInfo, buildContext)
    % Get file extensions for the current platform
    [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

    % Add file paths
    hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
    buildInfo.addIncludePaths(hdrFilePath);

    % Link files
    linkFiles = strcat('adder', linkLibExt);
    linkPath = hdrFilePath;
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    % Non-build files for packaging
    nbFiles = 'adder';
    nbFiles = strcat(nbFiles, execLibExt);
    buildInfo.addNonBuildFiles(nbFiles,'','');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % Add the required include statements to the generated function code
        coder.cinclude('adder.h');
        coder.cinclude('adder_initialize.h');
        coder.cinclude('adder_terminate.h');
        c = 0;
```

```
                % Because MATLAB Coder generated adder, use the
                % housekeeping functions before and after calling
                % adder with coder.ceval.

                coder.ceval('adder_initialize');
                c = coder.ceval('adder', a, b);
                coder.ceval('adder_terminate');
            end
        end
    end
end
```

Write a function `adder_main` that calls the external library function `adder`.

```
function y = adder_main(x1, x2)
    %#codegen
    y = AdderAPI.adder(x1, x2);
end
```

Generate a MEX function for `adder_main`. The MEX Function exercises the `coder.ExternalDependency` methods.

```
codegen('adder_main','-args',{7,9},'-report')
```

Copy the library to the current folder using the file extension for your platform. For Windows, use:

```
copyfile(fullfile(pwd,'codegen','dll','adder','adder.dll'));
```

For Linux, use:

```
copyfile(fullfile(pwd,'codegen','dll','adder','adder.so'));
```

Run the MEX function and verify the result.

```
adder_main_mex(2,3)
```

See Also

`coder.BuildConfig` | `coder.ceval` | `coder.cinclude` | `coder.updateBuildInfo`

Topics

“Develop Interface for External C/C++ Code”

“Build Process Customization”
“Integrate External/Custom Code”

Introduced in R2013b

coder.fftw.StandaloneFFTW3Interface class

Package: coder.fftw

Abstract class for specifying an FFTW library for FFTW calls in generated code

Description

`coder.fftw.StandaloneFFTW3Interface` is an abstract class for defining an FFT library callback class. An FFT library callback class specifies an FFT library to use for C/C++ code generated for MATLAB fast Fourier transform functions. To define an FFT callback class for the FFTW library, version 3.2 or later, use the `coder.fftw.StandaloneFFTW3Interface` class. For example, to define an FFT library callback class with the name `useMyFFTW`, make this line the first line of your class definition file:

```
classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface
```

For information about the FFTW library, see www.fftw.org.

MATLAB fast Fourier transform functions include `fft`, `fft2`, `fftn`, `ifft`, `ifft2`, and `ifftn`. The code generator produces FFTW library calls for these functions when all of these conditions are true:

- You generate standalone C/C++ code (static library, dynamically linked library, or executable program) with MATLAB Coder or generate C/C++ code from a MATLAB Function block with Simulink Coder.
- You have access to an FFTW library installation, version 3.2 or later.
- You specify the FFTW library installation in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.
- You set the appropriate configuration parameter to the name of the callback class.
  - For code generation with the MATLAB Coder `codegen` command, set `CustomFFTCallback`.
  - For code generation with the MATLAB Coder app, set **Custom FFT library callback**.
  - For code generation for a MATLAB Function block by using Simulink Coder, set **Custom FFT library callback**.

You must implement the `updateBuildInfo` and `getNumThreads` methods.

Optionally, you can implement these methods:

- `getPlanMethod`
- `lock` and `unlock`

All methods are static.

Methods

getNumThreads	Return number of threads to use for FFTW library calls
getPlanMethod	Return FFTW planning method
lock	Lock access to FFTW planning
unlock	Unlock access to FFTW planning
updateBuildInfo	Update the build information for linking to a specific FFTW library

Examples

Write an FFT Library Callback Class for an FFTW Library

Specify a specific installed FFTW library in an FFT library callback class.

Use this example FFT library callback class as a template.

```
% copyright 2017 The MathWorks, Inc.

classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '/usr/lib/fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
```

```
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
             libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);
        end
    end
end
```

Modify the template.

- Replace `useMyFFTW` with the name of your callback class.
- If your FFTW installation uses multiple threads, modify the `getNumThreads` method to return the number of threads that you want to use.
- In the `updateBuildInfo` method, set:
  - `fftwLocation` to the full path for your installation of the library.
  - `includePath` to the full path of the folder that contains `fftw3.h`.
  - `libPath` to the full path of the folder that contains the library files.

See Also

Topics

“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”
“Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code”
“Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)
“Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.fftw.org

Introduced in R2017b

coder.FiType class

Package: coder
Superclasses:

Represent set of MATLAB fixed-point arrays

Description

Specifies the set of fixed-point array values that the generated code should accept. Use only with the `codegen -args` options. Do not pass as an input to the generated MEX function.

Construction

`t=coder.typeof(v)` creates a `coder.FiType` object representing a set of fixed-point values whose properties are based on the fixed-point input `v`.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is [], the (upper bound) sizes of `v` do not change. If you do not specify the `variable_dims` input parameter, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to the bounded dimensions that are not `1` or `0` (which are fixed).

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims)` creates a `coder.Type` object representing a set of fixed-point values with `numerictype` and (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When you do not specify `variable_dims`, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to the bounded dimensions that are not `1` or `0` (which are fixed).

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)` creates a `coder.Type` object representing a set of fixed-point values with `numerictype` and additional options specified by one or more Name, Value pair

arguments. `Name` can also be a property name and `Value` is the corresponding value. Specify `Name` as a character vector or string scalar. You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

Input Arguments

`v`

Fixed-point value used to create new `coder.FiType` object.

`sz`

Size vector specifying each dimension of type object.

Default: [1 1] for `coder.newtype`

`variable_dims`

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

Default: false(size(sz)) | sz ==Inf for `coder.newtype`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`complex`

Set `complex` to `true` to create a `coder.Type` object that can represent complex values. The type must support complex data.

Default: false

`fimath`

Specify local `fimath`. If not, uses default `fimath`.

Properties

`ClassName`

Class of values in the set.

`Complex`

Indicates whether fixed-point arrays in the set are real (`false`) or complex (`true`).

`Fimath`

Local `fimath` that the fixed-point arrays in the set use.

`NumericType`

numerictype that the fixed-point arrays in the set use.

`SizeVector`

The upper-bound size of arrays in the set.

`VariableDims`

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a new fixed-point type `t`.

```
t = coder.typeof(fi(1));
% Returns
% coder.FiType
%   1x1 embedded.fi
%       DataTypeMode:Fixed-point: binary point scaling
```

```
%         Signedness:Signed
%         WordLength:16
%     FractionLength:14
```

Create a new fixed-point type for use in code generation. The fixed-point type uses the default `fimath`.

```
t = coder.newtype('embedded.fi',numerictype(1, 16, 15), [1 2])

t =
% Returns
% coder.FiType
%   1x2 embedded.fi
%           DataTypeMode: Fixed-point: binary point scaling
%           Signedness: Signed
%           WordLength: 16
%           FractionLength: 15
```

This new type uses the default `fimath`.

See Also

`codegen` | `coder.ArrayType` | `coder.ClassType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof`

Introduced in R2011a

coder.FixptConfig class

Package: coder

Floating-point to fixed-point conversion configuration object

Description

A `coder.FixptConfig` object contains the configuration parameters that the MATLAB Coder `codegen` function requires to convert floating-point MATLAB code to fixed-point MATLAB code during code generation. Use the `-float2fixed` option to pass this object to the `codegen` function.

Construction

`fixptcfg` `= coder.config('fixpt')` creates a `coder.FixptConfig` object for floating-point to fixed-point conversion.

Properties

`ComputeDerivedRanges`

Enable derived range analysis.

Values: `true`|`false` (default)

`ComputeSimulationRanges`

Enable collection and reporting of simulation range data. If you need to run a long simulation to cover the complete dynamic range of your design, consider disabling simulation range collection and running derived range analysis instead.

Values: `true` (default)|`false`

`DefaultFractionLength`

Default fixed-point fraction length.

Values: 4 (default) | positive integer

`DefaultSignedness`

Default signedness of variables in the generated code.

Values: `'Automatic'` (default) | `'Signed'` | `'Unsigned'`

`DefaultWordLength`

Default fixed-point word length.

Values: 14 (default) | positive integer

`DetectFixptOverflows`

Enable detection of overflows using scaled doubles.

Values: `true`| `false` (default)

`fimath`

`fimath` properties to use for conversion.

Values: `fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'SumMode', 'FullPrecision')` (default) | string

`FixPtFileNameSuffix`

Suffix for fixed-point file names.

Values: `'_fixpt'` | string

`LaunchNumericTypesReport`

View the numeric types report after the software has proposed fixed-point types.

Values: `true` (default) | `false`

`LogIOForComparisonPlotting`

Enable simulation data logging to plot the data differences introduced by fixed-point conversion.

Values: `true` (default) | `false`

`OptimizeWholeNumber`

Optimize the word lengths of variables whose simulation min/max logs indicate that they are always whole numbers.

Values: `true` (default) | `false`

`PlotFunction`

Name of function to use for comparison plots.

`LogIOForComparisonPlotting` must be set to true to enable comparison plotting. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

Values: `''` (default) | string

`PlotWithSimulationDataInspector`

Use Simulation Data Inspector for comparison plots.

`LogIOForComparisonPlotting` must be set to true to enable comparison plotting. The `PlotFunction` option takes precedence over `PlotWithSimulationDataInspector`.

Values: `true`| `false` (default)

`ProposeFractionLengthsForDefaultWordLength`

Propose fixed-point types based on `DefaultWordLength`.

Values: `true` (default) | `false`

`ProposeTargetContainerTypes`

By default (false), propose data types with the minimum word length needed to represent the value. When set to true, propose data type with the smallest word length that can

represent the range and is suitable for C code generation (8,16,32, 64 ...). For example, for a variable with range [0..7], propose a word length of 8 rather than 3.

Values: `true`| `false` (default)

`ProposeWordLengthsForDefaultFractionLength`

Propose fixed-point types based on `DefaultFractionLength`.

Values: `false` (default) | `true`

`ProposeTypesUsing`

Propose data types based on simulation range data, derived ranges, or both.

Values: `'BothSimulationAndDerivedRanges'` (default) | `'SimulationRanges'`|`'DerivedRanges'`

`SafetyMargin`

Safety margin percentage by which to increase the simulation range when proposing fixed-point types. The specified safety margin must be a real number greater than `-100`.

Values: 0 (default) | double

`StaticAnalysisQuickMode`

Perform faster static analysis.

Values: `true` | `false` (default)

`StaticAnalysisTimeoutMinutes`

Abort analysis if timeout is reached.

Values: `''` (default) | positive integer

`TestBenchName`

Test bench function name or names, specified as a string or cell array of strings. You must specify at least one test bench.

If you do not explicitly specify input parameter data types, the conversion uses the first test bench function to infer these data types.

Values: `''` (default) | string | cell array of strings

`TestNumerics`

Enable numerics testing.

Values: `true`| `false` (default)

Methods

addDesignRangeSpecification	Add design range specification to parameter
addFunctionReplacement	Replace floating-point function with fixed-point function during fixed-point conversion
clearDesignRangeSpecifications	Clear all design range specifications
getDesignRangeSpecification	Get design range specifications for parameter
hasDesignRangeSpecification	Determine whether parameter has design range
removeDesignRangeSpecification	Remove design range specification from parameter
addApproximation	Replace floating-point function with lookup table during fixed-point conversion

Examples

Generate Fixed-Point C Code from Floating-Point MATLAB Code

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Create a code generation configuration object to generate a standalone C static library.

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

Convert a floating-point MATLAB function to fixed-point C code. In this example, the MATLAB function name is `dti`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

Convert Floating-Point MATLAB Code to Fixed Point Based On Derived Ranges

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the name of the test bench to use to infer input data types. In this example, the test bench function name is `dti_test`. The conversion process uses the test bench to infer input data types.

```
fixptcfg.TestBenchName = 'dti_test';
```

Select to propose data types based on derived ranges.

```
fixptcfg.ProposeTypesUsing = 'DerivedRanges';
fixptcfg.ComputeDerivedRanges = true;
```

Add design ranges. In this example, the `dti` function has one scalar double input, `u_in`. Set the design minimum value for `u_in` to `-1` and the design maximum to `1`.

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
```

Convert the floating-point MATLAB function, `dti`, to fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg  dti
```

Enable Overflow Detection

When you select to detect potential overflows, `codegen` generates a scaled double version of the generated fixed-point MEX function. Scaled doubles store their data in double-precision floating-point, so they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type.

This example requires MATLAB Coder and Fixed-Point Designer licenses.

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Enable numerics testing with overflow detection.

```
fixptcfg.TestNumerics = true;
fixptcfg.DetectFixptOverflows = true;
```

Create a code generation configuration object to generate a standalone C static library.

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

Convert a floating-point MATLAB function to fixed-point C code. In this example, the MATLAB function name is `dti`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

Alternatives

You can convert floating-point MATLAB code to fixed-point code using the MATLAB Coder app. Open the app using one of these methods:

- On the **Apps** tab, in the **Code Generation** section, click **MATLAB Coder**.
- Use the `coder` command.

See “Convert MATLAB Code to Fixed-Point C Code”.

See Also

`codegen` | `coder` | `coder.CodeConfig` | `coder.config`

Topics

“Generate C Code at the Command Line”

coder.LAPACKCallback class

Package: coder

Abstract class for specifying the LAPACK library and LAPACKE header file for LAPACK calls in generated code

Description

`coder.LAPACKCallback` is an abstract class for defining a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file to use for LAPACK calls in code generated from MATLAB code. If you use MATLAB Coder to generate standalone code or generate code for the MATLAB Function block, for certain linear algebra function calls, you can generate LAPACK calls. To generate LAPACK calls, set the appropriate configuration parameter to the name of the LAPACK callback class.

- For code generation with MATLAB Coder `codegen` command, set `CustomLAPACKCallback`.
- For code generation with MATLAB Coder app, set **Custom LAPACK library callback**.
- For code generation for a MATLAB Function block with Simulink Coder, set **Custom LAPACK library callback**.

To define a LAPACK callback class with the name `useMyLAPACK`, make the following line the first line of your class definition file.

```
classdef useMyLAPACK < coder.LAPACKCallback
```

You must define all of the methods listed in “Methods” on page 3-49. These methods are static and are not compiled.

Methods

getHeaderFilename	Return file name of LAPACKE header file
updateBuildInfo	Update build information for linking to a specific LAPACK library

Examples

Write a LAPACK Callback Class

This example shows how to write a LAPACK callback class.

Use this example LAPACK callback class as a template.

```
classdef useMyLAPACK < coder.LAPACKCallback
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
        function updateBuildInfo(buildInfo, buildctx)
            buildInfo.addIncludePaths(fullfile(pwd,'include'));
            libName = 'mylapack';
            libPath = fullfile(pwd,'lib');
            [~,linkLibExt] = buildctx.getStdLibInfo();
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
                '', true, true);
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
        end
    end
end
```

Replace `useMyLAPACK` with the name of your callback class.

The `getHeaderFilename` method returns the name of the header file for the LAPACKE C interface to the LAPACK library. Replace `mylapacke_custom.h` with the name of your LAPACKE header file.

The `updateBuildInfo` method updates the build information with the locations of the header files and the name and location of the LAPACK library. Replace `mylapack` with the name of your LAPACK library.

If your compiler supports only complex data types that are represented as structures, include these lines in the `updateBuildInfo` method.

```
buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
```

See Also

`coder.BuildConfig` | `coder.ExternalDependency`

Topics

“Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”
“Speed Up Linear Algebra in Code Generated from a MATLAB Function Block” (Simulink Coder)

External Websites

www.netlib.org/lapack

Introduced in R2016a

coder.BLASCallback class

Package: coder

Abstract class for specifying the BLAS library and CBLAS header and data type information for BLAS calls in generated code

Description

`coder.BLASCallback` is an abstract class for defining a BLAS callback class. A BLAS callback class specifies the BLAS library and CBLAS header and data type information to use for BLAS calls in code generated from MATLAB code. If you use MATLAB Coder to generate standalone code or generate code for the MATLAB Function block, for certain vector and matrix function calls, you can generate BLAS calls. To generate BLAS calls, set the appropriate configuration parameter to the name of the BLAS callback class.

- For code generation by using the MATLAB Coder `codegen` command, set `CustomBLASCallback`.
- For code generation by using the MATLAB Coder app, set **Custom BLAS library callback**.
- For code generation for a MATLAB Function block by using Simulink Coder, set **Custom BLAS library callback**.

To define a BLAS callback class with the name `useMyBLAS`, make the following line the first line of your class definition file.

```
classdef useMyBLAS < coder.BLASCallback
```

You must define the `updateBuildInfo`, `getHeaderFileName`, and `getBLASIntTypeName` methods. The other methods, `getBLASDoubleComplexTypeName`, `getBLASSingleComplexTypeName`, and `useEnumNameRatherThanTypedef`, are already implemented in `coder.BLASCallback`. In certain situations, you must override these methods with your own definitions when you define your callback class. All methods are static and are not compiled.

Class Attributes

Abstract

true

For information on class attributes, see “Class Attributes” (MATLAB).

Methods

Public Methods

<infotypegroup type="method"> `updateBuildInfo` `getHeaderFilename` `getBLASIntTypeName` `getBLASDoubleComplexTypeName` `getBLASSingleComplexTypeName` `useEnumNameRatherThanTypedef` </infotypegroup>

Examples

Callback Class for Intel MKL BLAS

This example is an implementation of the BLAS callback class `mklcallback` for linking to the Intel MKL BLAS library on a Windows platform. `mklcallback` does not include explicit implementations of `getBLASDoubleComplexTypeName`, `getBLASSingleComplexTypeName`, and `useEnumNameRatherThanTypedef`. It inherits these methods from `coder.BLASCallback`.

```
classdef mklcallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, ~)
            libPath = fullfile(pwd,'mkl','WIN','lib','intel64');
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libs = {'mkl_intel_ilp64.lib' 'mkl_intel_thread.lib' 'mkl_core.lib'};
            buildInfo.addLinkObjects(libs, libPath, libPriority, libPreCompiled, libLi
            buildInfo.addLinkObjects('libiomp5md.lib',fullfile(matlabroot,'bin','win64
                libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(fullfile(pwd,'mkl','WIN','include'));
```

```
            buildInfo.addDefines('-DMKL_ILP64');
        end
        function headerName = getHeaderFilename()
            headerName = 'mkl_cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
            intTypeName = 'MKL_INT';
        end
    end
end
```

Use this example class as a template for writing your own BLAS callback class.

If you are using a different BLAS library, replace `'mkl_cblas.h'` with the name of your CBLAS header file.

If you are using a different BLAS library, replace `'MKL_INT'` with the name of your CBLAS integer data type.

To update the build information in `updateBuildInfo` with the name and location of your BLAS library, use the build information `addLinkObjects` method. If you use the Intel MKL BLAS library, use the link line advisor to see which libraries and compiler options are recommended for your use case.

To update the build information in `updateBuildInfo` with the location of the CBLAS header files, use the build information `addIncludePaths` method.

To add preprocessor macro definitions to the build information in `updateBuildInfo`, use the build information `addDefines` method.

To specify the compiler options in `updateBuildInfo`, use the build information `addCompileFlags` method.

To specify the linker options in `updateBuildInfo`, use the build information `addLinkFlags` method.

The `getBLASDoubleComplexTypeName` method returns the type used for double-precision complex variables in the generated code. If your BLAS library takes a type other than `double*` and `void*` for double-precision complex array arguments, include this method in your callback class definition.

```
function doubleComplexTypeName = getBLASDoubleComplexTypeName()
doubleComplexTypeName = 'my_double_complex_type';
end
```

Replace `my_double_complex_type` with the type that your BLAS library takes for double-precision complex array arguments.

The `getBLASSingleComplexTypeName` method returns the type used for single-precision complex variables in the generated code. If your BLAS library takes a type other than `float*` and `void*` for single-precision complex array arguments, include this method in your callback class definition.

```
function singleComplexTypeName = getBLASSingleComplexTypeName()
doubleComplexTypeName = 'my_single_complex_type';
end
```

Replace `my_single_complex_type` with the type that your BLAS library takes for single-precision complex array arguments.

The `useEnumNameRatherThanTypedef` method returns `false` by default. If types for enumerations in your BLAS library include the `enum` keyword, redefine this method to return `true` in your callback class definition.

```
function p = useEnumNameRatherThanTypedef()
p = true;
end
```

Callback Class for OpenBLAS

This example is an implementation of the BLAS callback class `openblascallback` for linking to the OpenBLAS library on a Linux platform. `openblascallback` does not have explicit implementations of `getBLASDoubleComplexTypeName`, `getBLASSingleComplexTypeName`, and `useEnumNameRatherThanTypedef`. It inherits these methods from `coder.BLASCallback`.

```
classdef openblascallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, buildctx)
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libName = 'libopenblas.a';
            libPath = fullfile(pwd,'openblas');
            incPath = fullfile(pwd,'openblas');
            buildInfo.addLinkFlags('-lpthread');
            buildInfo.addLinkObjects(libName, libPath, ...
```

```
                libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(incPath);
        end
        function headerName = getHeaderFilename()
            headerName = 'cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
            intTypeName = 'blasint';
        end
    end
end
```

If you generate C++ code that includes calls to OpenBLAS library functions, compiling it with the `-pedantic` option produces warnings. To disable the `-pedantic` compiler option, in the `updateBuildInfo` method, include these lines:

```
if buildctx.getTargetLang() == 'C++'
    buildInfo.addCompileFlags('-Wno-pedantic');
end
```

OpenBLAS does not support the C89/C90 standard.

See Also

`coder.BuildConfig` | `coder.ExternalDependency`

Topics

“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
“Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder)
“Build Process Customization”

External Websites

http://www.netlib.org/blas/
http://www.openblas.net/
https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines
https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

Introduced in R2018b

coder.PrimitiveType class

Package: coder
Superclasses:

Represent set of logical, numeric, or char arrays

Description

Specifies the set of logical, numeric, or char values that the generated code should accept. Supported classes are `double`,`single`,`int8`,`uint8`,`int16`,`uint16`,`int32`,`uint32`,`int64`,`uint64`, `char`, and `logical`. Use only with the `codegen -args` option. Do not pass as an input to a generated MEX function.

Construction

`t=coder.typeof(v)` creates a `coder.PrimitiveType` object denoting the smallest non-constant type that contains `v`. `v` must be a MATLAB numeric, logical or char.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `sz` is [], the (upper bound) sizes of `v` remain unchanged. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to bounded dimensions that are not `1` or `0` (which are assumed to be fixed).

`t=coder.newtype(numeric_class, sz, variable_dims)` creates a `coder.PrimitiveType` object representing values of class `numeric_class` with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the dimensions of the type that are not `1` or `0` (which are assumed to be fixed).

`t=coder.newtype(numeric_class, sz, variable_dims, Name, Value)` creates a `coder.PrimitiveType` object with additional options specified by one or more Name, Value pair arguments. `Name` can also be a property name and `Value` is the corresponding value. Specify `Name` as character vector or string scalar. You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

Input Arguments

`v`

Input that is not a `coder.Type` object

`sz`

Size for corresponding dimension of type object. Size must be a valid size vector.

Default: [1 1] for `coder.newtype`

`variable_dims`

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

Default: false(size(sz)) | sz==Inf for `coder.newtype`

`numeric_class`

Class of type object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`complex`

Set `complex` to `true` to create a `coder.PrimitiveType` object that can represent complex values. The type must support complex data.

Default: false

`sparse`

Set `sparse` to `true` to create a `coder.PrimitiveType` object representing sparse data. The type must support sparse data.

Default: false

Properties

`ClassName`

Class of values in this set

`Complex`

Indicates whether the values in this set are real (`false`) or complex (`true`)

`SizeVector`

The upper-bound size of arrays in this set.

`Sparse`

Indicates whether the values in this set are sparse arrays (`true`)

`VariableDims`

A vector used to specify whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a `coder.PrimitiveType` object.

```
z = coder.typeof(0,[2 3 4],[1 1 0]) % returns double :2x:3x4
% ':' indicates variable-size dimensions
```

Create a `coder.PrimitiveType` object then call `codegen` to generate a C library for a function `fcn.m` that has one input parameter of this type.

1. Create a `coder.PrimitiveType` object.

   ```
   z = coder.typeof(0,[2 3 4],[1 1 0]) % returns double :2x:3x4
   % ':' indicates variable-size dimensions
   ```

2. Call `codegen` to generate a C library for a MATLAB function `fcn.m` that has one input parameter type `z`.

   ```
   % Use the config:lib option to generate a C library
   codegen -config:lib fcn -args {z}
   ```

See Also

`codegen` | `coder.ArrayType` | `coder.ClassType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof`

Introduced in R2011a

coder.StructType class

Package: coder
Superclasses:

Represent set of MATLAB structure arrays

Description

Specifies the set of structure arrays that the generated code should accept. Use only with the `codegen -args` option. Do not pass as an input to a generated MEX function.

Construction

`t=coder.typeof(struct_v)` creates a `coder.StructType` object for a structure with the same fields as the scalar structure `struct_v`.

`t=coder.typeof(struct_v, sz, variable_dims)` returns a modified copy of `coder.typeof(struct_v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `sz` is [], the (upper bound) sizes of `struct_v` remain unchanged. If the `variable_dims` input parameter is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the bounded dimensions that are not `1` or `0` (which are assumed to be fixed).

`t=coder.newtype('struct', struct_v, sz, variable_dims)` creates a `coder.StructType` object for an array of structures with the same fields as the scalar structure `struct_v` and (upper bound) size `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the dimensions of the type, except if the dimension is `1` or `0`, which is assumed to be fixed.

Input Arguments

`struct_v`

Scalar structure used to specify the fields in a new structure type.

`sz`

Size vector specifying each dimension of type object.

Default: [1 1] for `coder.newtype`

`variable_dims`

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

Default: false(size(sz)) | sz==Inf for `coder.newtype`

Properties

`Alignment`

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary so it will not be matched by CRL functions that require alignment.

`Alignment` must be either `-1` or a power of `2` that is no more than `128`.

`ClassName`

Class of values in this set.

`Extern`

Whether the structure type is externally defined.

`Fields`

A structure giving the `coder.Type` of each field in the structure.

`HeaderFile`

If the structure type is externally defined, name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`. Specify the path to the file using the `codegen -I` option or the **Additional include directories** parameter in the MATLAB Coder project settings dialog box **Custom Code** tab.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty character vector or string scalar.

`SizeVector`

The upper-bound size of arrays in this set.

`VariableDims`

A vector used to specify whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a type for a structure with a variable-size field.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
% Returns
% coder.StructType
```

```
%    1x1 struct
%      a:  :3x:5 double
%      b:  3x3  double
% ':' indicates variable-size dimensions
```

Create a `coder.StructType` object then call `codegen` to generate a C library for a function `fcn.m` that has one input parameter of this type

1. Create a new structure type.

   ```
   ta = coder.newtype('int8',[1 1]);
   tb = coder.newtype('double',[1 2],[1 1]);
   z = coder.newtype('struct',struct('a',ta,'b',tb))
   % Returns
   % coder.StructType
   %   1x1 struct
   %      a: 1x1 int8
   %      b: :1x:2 double
   ```

2. Call `codegen` to generate a C library for a MATLAB function `fcn.m` that has one input parameter of this type.

   ```
   % Use the -config:lib option to generate a C library
   codegen -config:lib fcn -args {z}
   ```

Create a `coder.StructType` object that uses an externally-defined structure type.

1. Create a type that uses an externally-defined structure type.

   ```
   S.a = coder.typeof(double(0));
   S.b = coder.typeof(single(0));
   T = coder.typeof(S);
   T = coder.cstructname(T,'mytype','extern','HeaderFile','myheader.h');
   ```

   ```
   T =

   coder.StructType
      1x1 extern mytype (myheader.h) struct
         a: 1x1 double
         b: 1x1 single
   ```

2. View the types of the structure fields.

   ```
   T.Fields
   ```

   ```
   ans =
   ```

```
    a: [1x1 coder.PrimitiveType]
    b: [1x1 coder.PrimitiveType]
```

See Also

codegen | coder | coder.ArrayType | coder.ClassType | coder.Constant | coder.EnumType | coder.FiType | coder.PrimitiveType | coder.Type | coder.cstructname | coder.newtype | coder.resize | coder.typeof

Introduced in R2011a

coder.SingleConfig class

Package: coder

Double-precision to single-precision conversion configuration object

Description

A `coder.SingleConfig` object contains the configuration parameters that the MATLAB Coder `codegen` function requires to convert double-precision code to single-precision MATLAB code. To pass this object to the `codegen` function, use the `-double2single` option.

Construction

`scfg` `= coder.config('single')` creates a `coder.SingleConfig` object for double-precision to single-precision conversion.

Properties

`OutputFileNameSuffix` — Suffix for single-precision file name
`'_single'` (default) | character vector

Suffix that the single-conversion process uses for generated single-precision files.

`LogIOForComparisonPlotting` — Enable simulation data logging for comparison plotting of input and output variables
`false` (default) | true

Enable simulation data logging to plot the data differences introduced by single-precision conversion.

`PlotFunction` — Name of function for comparison plots
`''` (default) | character vector

Name of function to use for comparison plots.

To enable comparison plotting, set `LogIOForComparisonPlotting` to true. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function must accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

`PlotWithSimulationDataInspector` — Specify use of Simulation Data Inspector for comparison plots
`false` (default) | `true`

Use Simulation Data Inspector for comparison plots.

`LogIOForComparisonPlotting` must be set to true to enable comparison plotting. The `PlotFunction` option takes precedence over `PlotWithSimulationDataInspector`.

`TestBenchName` — Name of test file
`''` (default) | character vector | cell array of character vectors

Test file name or names, specified as a character vector or cell array of character vectors. Specify at least one test file.

If you do not explicitly specify input parameter data types, the conversion uses the first file to infer these data types.

`TestNumerics` — Enable numerics testing
`false` (default) | `true`

Enable numerics testing to verify the generated single-precision code. The test file runs the single-precision code.

Methods

addFunctionReplacement	Replace double-precision function with single-precision function during single-precision conversion

Examples

Generate Single-Precision MATLAB Code

Create a `coder.SingleConfig` object.

```
scfg= coder.config('single');
```

Set the properties of the doubles-to-singles configuration object. Specify the test file. In this example, the name of the test file is `myfunction_test`. The conversion process uses the test file to infer input data types and collect simulation range data. Enable numerics testing and generation of comparison plots.

```
scfg.TestBenchName = 'myfunction_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

Run `codegen`. Use the `-double2single` option to specify the `coder.SingleConfig` that you want to use. In this example, the MATLAB function name is `myfunction`.

```
codegen -double2single scfg myfunction
```

Alternatives

You can convert double-precision MATLAB code to single-precision C/C++ code by using the `'singleC'` option of the `codegen` function.

You can convert double-precision MATLAB code to single-precision code using the MATLAB Coder app. Open the app using one of these methods:

- On the **Apps** tab, in the **Code Generation** section, click **MATLAB Coder**.
- Use the `coder` command.

See Also

`codegen` | `coder.config`

Topics

“Generate Single-Precision MATLAB Code”

“Generate Single-Precision C Code at the Command Line”

Introduced in R2015b

coder.Type class

Package: coder

Represent set of MATLAB values

Description

Specifies the set of values that the generated code should accept. Use only with the `codegen -args` option. Do not pass as an input to a generated MEX function.

Construction

`coder.Type` is an abstract class, and you cannot create instances of it directly. You can create `coder.Constant`, `coder.EnumType`, `coder.FiType`, `coder.PrimitiveType`, `coder.StructType`, and `coder.CellType` objects that are derived from this class.

Properties

`ClassName`

Class of values in this set

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

`codegen` | `coder` | `coder.ArrayType` | `coder.CellType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.newtype` | `coder.resize` | `coder.typeof`

Introduced in R2011a

coder.make.BuildConfiguration class

Package: coder.make

Represent build configuration

Description

A build configuration contains information on how to build source code and binaries.

Give each build configuration a unique name that you can use to reference or access it, such as ‘Faster Builds’.

A build configuration contains options with values. Each option maps to a build tool in the ToolchainInfo object that uses the build configuration.

For example, a build configuration can contain options for the following build tools in `coder.make.ToolchainInfo`:

- C Compiler
- C++ Compiler’
- Linker
- Shared Library Linker
- Archiver
- Download
- Execute

The value of each option can vary from one build configuration to another. For example, the “Faster Runs” build configuration can have compiler options that include optimization flags, while the “Debug” build configuration can have compiler options that include a symbolic debug flag.

Construction

```
ConfigObj = coder.make.BuildConfiguration(ConfigName,{Name,
Value,...})
```

Input Arguments

`ConfigName` — Name of build configuration

character vector

Name of build configuration, specified as a character vector.

Example: `'Faster Builds II'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Name` — Name of option

character vector

Name of option, specified as a character vector.

Data Types: `char`

`Value` — Value of option

character vector

Value of option, specified as a character vector.

Data Types: `char`

Output Arguments

`ConfigObj` — Object handle for configuration

variable

Object handle for configuration, returned as a variable.

Data Types: `char`

Properties

`Description` — Brief description of build configuration

A brief description of the build configuration. The MATLAB Coder software displays this description in the project build settings, on the **Hardware** tab, below the **Build Configuration** parameter.

You can assign a description to this property after you create the `BuildConfiguration` object.

```
config.Description = 'BldConfigDescription'
```

```
config =

##########################################
# Build Configuration : BldConfigName
# Description         : BldConfigDescription
##########################################
```

Data type: `char`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Name` — Name of build configuration

The name of the build configuration.

You can assign a name to this property when you create a `BuildConfiguration` object.

```
config = coder.make.BuildConfiguration ...
('BldConfigName',{'optiona','1','optionb','2','optionc','3'})
```

You can also assign a name to this property after you create a `BuildConfiguration` object.

```
config.Name = 'BldConfigName'
```

Both approaches produce the same result

```
config =

##########################################
```

```
# Build Configuration : BldConfigName
# Description         :
###############################################
```

Data type: `char`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Options` — List of options or settings for specific build configuration

A list of options or settings for a specific build configuration. This list contains name-value pairs. The `Options` property has an option for each `coder.make.BuildTool` object in `coder.make.Toolchain.BuildTools`. For example, `Options` has a `C Compiler` option for the `C Compiler` build tool.

Data type: `coder.make.UnorderedList`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Methods

addOption	Add new option
getOption	Get value of option
info	Get information about build configuration
isOption	Determine if option exists
keys	Get all option names
setOption	Set value of option
values	Get all option values

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

See Also

`coder.make.BuildItem` | `coder.make.BuildTool` | `coder.make.ToolchainInfo` | `getBuildConfiguration` | `removeBuildConfiguration` | `setBuildConfiguration` | `setBuildConfigurationOption`

Topics

“Adding a Custom Toolchain”
“Toolchain Definition File with Commentary”

coder.make.BuildItem class

Package: coder.make

Represent build item

Description

Create a `coder.make.BuildItem` object that can have macro name and value. Then, use the `BuildItem` object as an argument for one of the following `coder.make.BuildTool` methods:

- `coder.make.BuildTool.getCommand`
- `coder.make.BuildTool.setCommand`
- `coder.make.BuildTool.setPath`
- `coder.make.BuildTool.addFileExtension`

Note What is a macro? The term has a different meaning depending on the context:

- In this context, a macro is a variable that the makefile can use to refer to a given value, such as a build tool's command, path, or file extension.
- In topics for the `coder.make.ToolchainInfo.Macros` and related methods, a macro is a variable that the makefile can use to refer to arbitrary or predefined value.

Construction

`h = coder.make.BuildItem(blditm_macrovalue)` creates a `coder.make.BuildItem` object that has a value.

`h = coder.make.BuildItem(blditm_macroname,blditm_value)` creates a `coder.make.BuildItem` object that has a macro name and value.

Input Arguments

`blditm_macroname` — Macro name of build item
character vector

Macro name of build item, specified as a character vector.

Data Types: `char`

`blditm_value` — Value of build item
character vector

Value of build item

Data Types: `char`

Output Arguments

`buildItemHandle` — BuildItem handle
object handle

BuildItem handle, specified as a `coder.make.BuildItem` object that contains an option value.

Example: `bi`

Methods

getMacro	Get macro name of build item
getValue	Get value of build item
setMacro	Set macro name of build item
setValue	Set value of build item

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Example

```
bi1 = coder.make.BuildItem('BuildItemMacroValue')


bi1 =

    Macro  : (empty)
    Value : BuildItemMacroValue

bi2 = coder.make.BuildItem('BIMV','BuildItemMacroValue')

bi2 =

    Macro  : BIMV
    Value : BuildItemMacroValue
```

See Also

`coder.make.ToolchainInfo` | `coder.make.BuildTool` | `coder.make.ToolchainInfo` | `getCommand` | `setCommand` | `setPath` | `addFileExtension`

Topics

“Adding a Custom Toolchain”

coder.make.BuildTool class

Package: coder.make

Represent build tool

Description

Use `coder.make.BuildTool` to get and define an existing default `coder.make.BuildTool` object, or to create a `coder.make.BuildTool` object.

To work with default `BuildTool` objects, use the get and define approach from the `ToolchainInfo` properties:

- `coder.make.ToolchainInfo.BuildTools`
- `coder.make.ToolchainInfo.PostbuildTools`

Examples showing the get and define approach are:

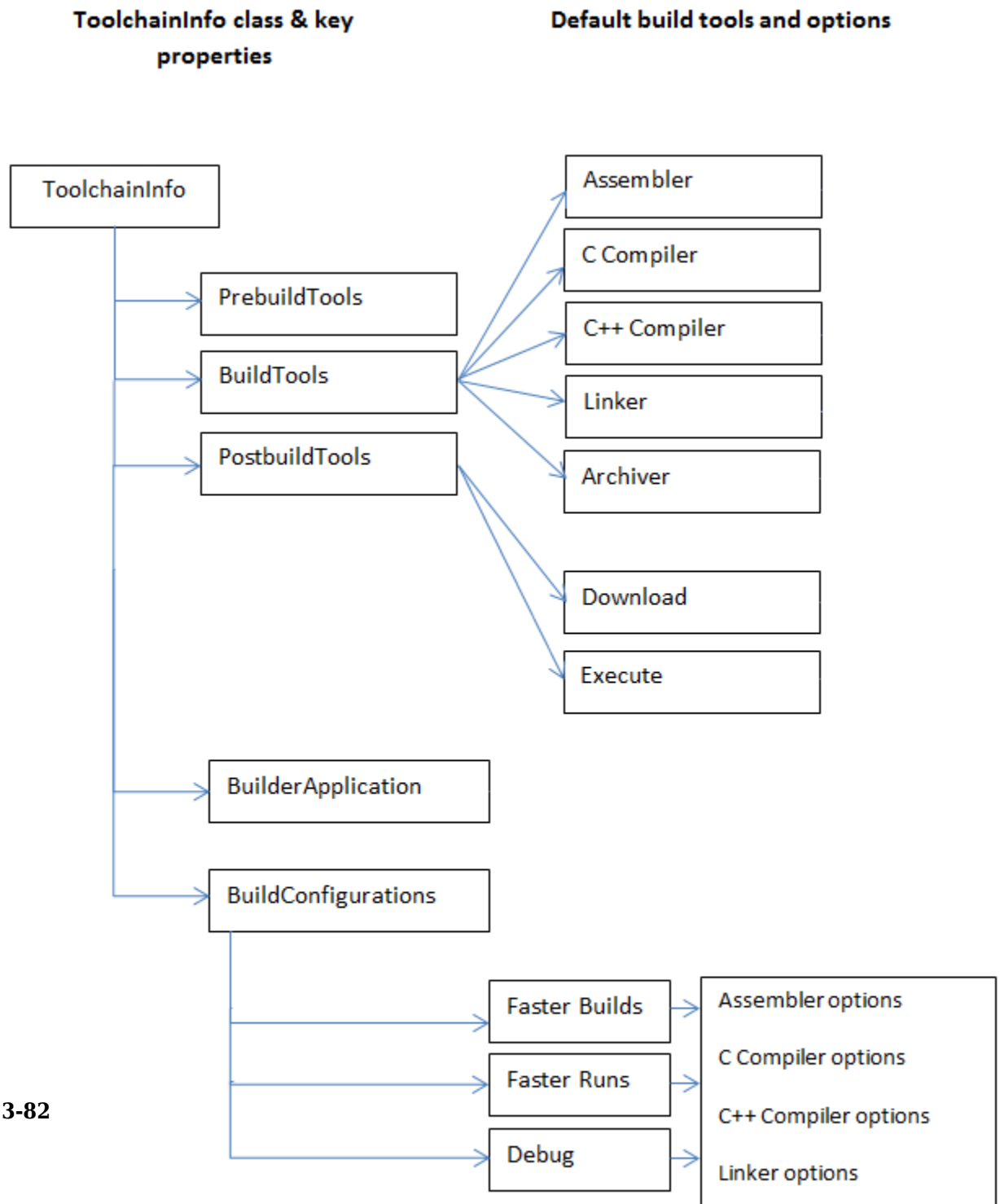
- "Toolchain Definition File with Commentary"
- Tutorial example: "Adding a Custom Toolchain" tutorial

An alternative to the get and define approach is the create new approach. An example showing the create new approach appears in "Create a Non-Default BuildTool" on page 3-88.

The illustration shows the relationship between the default `BuildTool` objects and `ToolchainInfo`. When you examine the `PHONY TARGETS` section of the generated makefile, the difference between the `BuildTools`, `PostbuildTools`, and `PrebuildTools` becomes clearer.

- `prebuild` – runs only the prebuild tool.
- `build` – runs the build tools after running prebuild. The `build` generates the build `PRODUCT`.
- `postbuild` – runs the postbuild tool after running build.
- `all` – runs prebuild, build, and postbuild. The build process uses this rule on a Ctrl+B build.

- `download` – is an empty rule by default.
- `execute` – is an empty rule by default.
- `clean` – cleans up all output file extensions and derived file extensions of all the tools in the toolchain.
- `info` – expands and prints all macros used in the makefile.


ToolchainInfo class & key properties
Default build tools and options
ToolchainInfo
PrebuildTools
BuildTools
PostbuildTools
BuilderApplication
BuildConfigurations
Assembler
C Compiler
C++ Compiler
Linker
Archiver
Download
Execute
Faster Builds
Faster Runs
Debug
Assembler options
C Compiler options
C++ Compiler options
Linker options

Construction

`h = coder.make.BuildTool(bldtl_name)` creates a `coder.make.BuildTool` object and sets its `Name` property.

Input Arguments

`bldtl_name` — Build tool name

character vector | string scalar

Build tool name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`h` — Object handle

variable

Object handle for a `coder.make.BuildTool` object, specified as a variable.

Example: `tool`

Properties

`Command` — Build tool command or command macro

Represents the build tool command using:

- An optional macro name, such as: `CC`.
- The system call (command) that starts the build tool, such as: `gcc`.

The macro name and system call appear together in the generated makefile. For example: `CC = gcc`

Assigning a value to this property is optional.

You can use the following methods with `Command`:

- `getCommand`
- `setCommand`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Directives` — Tool-specific directives

Defines any tool-specific directives, such as `-D` for preprocessor defines. Assigning a value to this property is optional.

You can use the following methods with `Directives`:

- `addDirective`
- `getDirective`
- `setDirective`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`FileExtensions` — Tool-specific file extensions

Defines any tool-specific file extensions. This value is optional.

You can use the following methods with `FileExtensions`:

- `addFileExtension`
- `getFileExtension`
- `setFileExtension`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Name` — Name of build tool

Defines the name of the build tool.

You can use the following methods with `Name`.

- `getName`
- `setName`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Path` — Tool-specific paths

Defines any tool-specific paths. If the command is on the system path, this value is optional.

You can use the following methods with `Path`:

- `getPath`
- `setPath`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`SupportedOutputs` — Tool-specific output formats

Defines any tool-specific output formats. If the tool supports all available formats, this value is optional.

The default value is `{'*'}`, which indicates support for all available formats.

The datatype is cell array. The contents of the cell array must be either `coder.make.BuildOutput` enumeration values, or `'*'`.

This property does not have any associated methods. Assign the value directly to the `SupportedOutputs`. See the `addPrebuildToolToToolchainInfo.m` example or the `addPostbuildToolToToolchainInfo.m` example. Valid enumeration values are:

`coder.make.BuildOutput.STATIC_LIBRARY`	applies for pre-build tools, build tools, and post-build tools
`coder.make.BuildOutput.SHARED_LIBRARY`	applies for build tools and post-build tools; requires Embedded Coder license
`coder.make.BuildOutput.EXECUTABLE`	applies for build tools and post-build tools

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Methods

addDirective	Add directive to `Directives`
addFileExtension	Add new file extension entry to `FileExtensions`
getCommand	Get build tool command
getDirective	Get value of named directive from `Directives`
getFileExtension	Get file extension for named file type in `FileExtensions`
getName	Get build tool name
getPath	Get path and macro of build tool in `Path`
info	Display build tool properties and values
setCommand	Set build tool command
setCommandPattern	Set pattern of commands for build tools
setCompilerOptionMap	Set C/C++ language standard and compiler options for selected build tool (compiler)
setDirective	Set value of directive in `Directives`
setFileExtension	Set file extension for named file type in `FileExtensions`
setName	Set build tool name
setPath	Set path and macro of build tool in `Path`
validate	Validate build tool properties

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Example

- “Get a Default Build Tool and Set Its Properties” on page 3-87
- “Create a Non-Default BuildTool” on page 3-88
- “Add Prebuild and Postbuild Tools to Toolchain” on page 3-92

Get a Default Build Tool and Set Its Properties

The `intel_tc.m` file from “Adding a Custom Toolchain” uses the following lines to get a default build tool, `C Compiler`, from a `ToolchainInfo` object called `tc`, and then sets its properties.

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

The following examples show the same “get and define” approach in more detail:

- “Toolchain Definition File with Commentary”
- Tutorial example: “Adding a Custom Toolchain” tutorial

Create a Non-Default BuildTool

To create a build tool:

1. Create a file that defines a `BuildTool` object, such as `createBuildTool_1.m` or `createBuildTool_2`.
2. Create a file like `addBuildToolToToolchainInfo.m`, that:
   - Creates a `ToolchainInfo` object, or uses an existing one.
   - Creates a `BuildTool` object from `createBuildTool_1.m` or `createBuildTool_2`.
   - Adds the `BuildTool` object to the `ToolchainInfo` object.
3. Run `addBuildToolToToolchainInfo.m`.

Refer to the following examples of `addBuildToolToToolchainInfo.m`, `createBuildTool_1.m`, and `createBuildTool_2.m`.

addBuildToolToToolchainInfo.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding a build tool to ToolchainInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Create a toolchain object
h = coder.make.ToolchainInfo();

% User function for creating and populating a build tool
tool = createBuildTool_1();
% or tool = createBuildTool_2();

% Add the build tool to ToolchainInfo
h.addBuildTool('My C Compiler',tool);
```

createBuildTool_1.m

```
function buildToolObj = createBuildTool_1()

toolinfo.Name                       = 'My GNU C Compiler';
toolinfo.Language                   = 'C';

toolinfo.Command.Macro              = 'CC';
toolinfo.Command.Value              = 'gcc';

toolinfo.Path.Macro                 = 'CC_PATH';
toolinfo.Path.Value                 = '';
```

```
toolinfo.OptionsRegistry            = {'My C Compiler','MY_CFLAGS'};

% Key name of this directive
toolinfo.Directives(1).Key          = 'IncludeSearchPath';

% Macro of this directive (directives can have empty macros)
toolinfo.Directives(1).Macro        = '';

% Value of this directive
toolinfo.Directives(1).Value        = '-I';

toolinfo.Directives(2).Key          = 'PreprocessorDefine';
toolinfo.Directives(2).Macro        = '';
toolinfo.Directives(2).Value        = '-D';

toolinfo.Directives(3).Key          = 'Debug';
toolinfo.Directives(3).Macro        = 'CDEBUG';
toolinfo.Directives(3).Value        = '-g';

toolinfo.Directives(4).Key          = 'OutputFlag';
toolinfo.Directives(4).Macro        = 'C_OUTPUT_FLAG';
toolinfo.Directives(4).Value        = '-o';

% Key name of this file extension
toolinfo.FileExtensions(1).Key      = 'Source';

% Macro of this file extension
toolinfo.FileExtensions(1).Macro    = 'C_EXT';

% Value of this file extension
toolinfo.FileExtensions(1).Value    = '.c';

toolinfo.FileExtensions(2).Key      = 'Header';
toolinfo.FileExtensions(2).Macro    = 'H_EXT';
toolinfo.FileExtensions(2).Value    = '.h';

toolinfo.FileExtensions(3).Key      = 'Object';
toolinfo.FileExtensions(3).Macro    = 'OBJ_EXT';
toolinfo.FileExtensions(3).Value    = '.obj';

toolinfo.DerivedFileExtensions      = {'$(OBJ_EXT)'};
% '*' means all outputs are supported
toolinfo.SupportedOutputs           = {'*'};


% put actual extension (e.g. '.c') or keyname if already registered
% under 'FileExtensions'
toolinfo.InputFileExtensions        = {'Source'};
toolinfo.OutputFileExtensions       = {'Object'};

% Create a build tool object and populate it with the above data
buildToolObj = createAndpopulateBuildTool(toolinfo);

function buildToolObj = createAndpopulateBuildTool(toolinfo)

% -------------------------
% Construct a BuildTool
% -------------------------
buildToolObj = coder.make.BuildTool();

% -------------------------
% Set general properties
```

```
% -------------------------
buildToolObj.Name              = toolinfo.Name;
buildToolObj.Language          = toolinfo.Language;
buildToolObj.Command           = coder.make.BuildItem ...
    (toolinfo.Command.Macro,toolinfo.Command.Value);
buildToolObj.Path              = coder.make.BuildItem ...
    (toolinfo.Path.Macro,toolinfo.Path.Value);
buildToolObj.OptionsRegistry   = toolinfo.OptionsRegistry;
buildToolObj.SupportedOutputs  = toolinfo.SupportedOutputs;

% -------------------------
% Directives
% -------------------------
for i = 1:numel(toolinfo.Directives)
    directiveBuildItem = coder.make.BuildItem(...
        toolinfo.Directives(i).Macro,toolinfo.Directives(i).Value);
    buildToolObj.addDirective(toolinfo.Directives(i).Key,directiveBuildItem);
end

% -------------------------
% File extensions
% -------------------------
for i = 1:numel(toolinfo.FileExtensions)
    fileExtBuildItem = coder.make.BuildItem(...
        toolinfo.FileExtensions(i).Macro,toolinfo.FileExtensions(i).Value);
    buildToolObj.addFileExtension(toolinfo.FileExtensions(i).Key,fileExtBuildItem);
end

% -------------------------
% Derived file extensions
% -------------------------
for i = 1:numel(toolinfo.DerivedFileExtensions)
    if buildToolObj.FileExtensions.isKey(toolinfo.DerivedFileExtensions{i})
        buildToolObj.DerivedFileExtensions{end+1} = ...
        ['$(' buildToolObj.getFileExtension
        (toolinfo.DerivedFileExtensions{i}) ')'];
    else
        buildToolObj.DerivedFileExtensions{end+1} = toolinfo.DerivedFileExtensions{i};
    end
end

% -------------------------
% Command pattern
% -------------------------
if isfield(toolinfo,'CommandPattern')
    buildToolObj.CommandPattern = toolinfo.CommandPattern;
end

% --------------------------------
% [Input/Output]FileExtensions
% --------------------------------
if isfield(toolinfo,'InputFileExtensions')
    buildToolObj.InputFileExtensions = toolinfo.InputFileExtensions;
end
if isfield(toolinfo,'OutputFileExtensions')
    buildToolObj.OutputFileExtensions = toolinfo.OutputFileExtensions;
end
```

createBuildTool_2.m

```
function buildToolObj = createBuildTool_2()
```

```
% -------------------------
% Construct a BuildTool
% -------------------------
buildToolObj = coder.make.BuildTool();

% -------------------------
% Set general properties
% -------------------------
buildToolObj.Name               = 'My GNU C Compiler';
buildToolObj.Language           = 'C';
buildToolObj.Command            = coder.make.BuildItem('CC','gcc');
buildToolObj.Path               = coder.make.BuildItem('CC_PATH','');
buildToolObj.OptionsRegistry    = {'My C Compiler','MY_CFLAGS'};
buildToolObj.SupportedOutputs   = {'*'}; % '*' means all outputs are supported

% -------------------------
% Directives
% -------------------------

directiveBuildItem = coder.make.BuildItem('','-I');
buildToolObj.addDirective('IncludeSearchPath',directiveBuildItem);

directiveBuildItem = coder.make.BuildItem('','-D');
buildToolObj.addDirective('PreprocessorDefine',directiveBuildItem);

directiveBuildItem = coder.make.BuildItem('CDEBUG','-g');
buildToolObj.addDirective('Debug',directiveBuildItem);

directiveBuildItem = coder.make.BuildItem('C_OUTPUT_FLAG','-o');
buildToolObj.addDirective('OutputFlag',directiveBuildItem);

% -------------------------
% File Extensions
% -------------------------

fileExtBuildItem = coder.make.BuildItem('C_EXT','.c');
buildToolObj.addFileExtension('Source',fileExtBuildItem);

fileExtBuildItem = coder.make.BuildItem('H_EXT','.h');
buildToolObj.addFileExtension('Header',fileExtBuildItem);

fileExtBuildItem = coder.make.BuildItem('OBJ_EXT','.obj');
buildToolObj.addFileExtension('Object',fileExtBuildItem);

% -------------------------
% Others
% -------------------------

buildToolObj.DerivedFileExtensions  = {'$(OBJ_EXT)'};
buildToolObj.InputFileExtensions    = {'Source'};
% put actual extension (e.g. '.c')
% or keyname if already registered under 'FileExtensions'
buildToolObj.OutputFileExtensions   = {'Object'};
% put actual extension (e.g. '.c')
% or keyname if already registered under 'FileExtensions'
```

Add Prebuild and Postbuild Tools to Toolchain

The code in the `addPrebuildToolToToolchainInfo.m` and the `addPostbuildToolToToolchainInfo.m` examples show how to add prebuild and postbuild tools to a toolchain.

addPrebuildToolToToolchainInfo.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding a pre-build tool with selected SupportedOutputs to ToolchainInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Create a toolchain object
tc = coder.make.ToolchainInfo(coder.make.getToolchainInfoFromRegistry(coder.make.getDefaultToolchain));
% Set inlined commands for source to dependency tool
tc.InlinedCommands = ['define sourceToDep=', 10, ...
'$(foreach source, $(1), $(CC) $(CFLAGS) -E -MMD -MP -MF"$(notdir $(source:%.c=%.d))" -MT"$(notdir $(source:
'endef'];
% Set makefile includes
make = tc.BuilderApplication();
make.IncludeFiles = {'*.d'};
% Dependency File Generator for GCC-based toolchain
prebuildToolName = 'Dependency File Generator';
prebuildTool = tc.addPrebuildTool(prebuildToolName);
% Set command macro and value
prebuildTool.setCommand('SRC2DEP', '$(call sourceToDep, $(SRCS))');
% Set tool options macro
prebuildTool.OptionsRegistry = {prebuildToolName, 'SRC2DEP_OPTS'};
% Set output type from tool
prebuildTool.SupportedOutputs = {'*'};
tc.addBuildConfigurationOption(prebuildToolName, prebuildTool);
tc.setBuildConfigurationOption('all', prebuildToolName, '');

% displays pre-build tool properties
tc.getPrebuildTool('Dependency File Generator')
```

addPostbuildToolToToolchainInfo.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding a post-build tool to ToolchainInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% ELF (executable and linkable format) to hexadecimal converter
postbuildToolName = 'elf2hex converter';
% Create and populate a post-build tool
tc = coder.make.ToolchainInfo;
postbuild = tc.addPostbuildTool(postbuildToolName);
% Set command macro and value for tool
postbuild.setCommand('OBJCOPY', 'arm-none-eabi-objcopy');
% Set path for tool
postbuild.setPath('OBJCOPYPATH','$(MW_GNU_ARM_TOOLS_PATH)');
% Set options for tool
postbuild.OptionsRegistry = {postbuildToolName, 'OBJCOPYFLAGS_HEX'};
```

```
% Set output type from tool
postbuild.SupportedOutputs = {coder.make.enum.BuildOutput.EXECUTABLE};
% Create build configuration for tool
tc.addBuildConfigurationOption(postbuildToolName, postbuild);
% Set build configuration for tool
tc.setBuildConfigurationOption('all', postbuildToolName, '-O ihex $(PRODUCT) $(PRODUCT_HEX)');

% displays post-build tool properties
tc.getPostbuildTool('elf2hex converter')
```

See Also

`coder.make.BuildTool` | `addBuildTool` | `getBuildTool` | `removeBuildTool` | `setBuildTool` | `coder.make.ToolchainInfo`

Topics

“Toolchain Definition File with Commentary”
“Adding a Custom Toolchain”
“About coder.make.ToolchainInfo”

coder.make.ToolchainInfo class

Package: coder.make

Represent custom toolchain

Description

Use `coder.make.ToolchainInfo` to define and register a new set of software build tools (toolchain) with MathWorks code generation products.

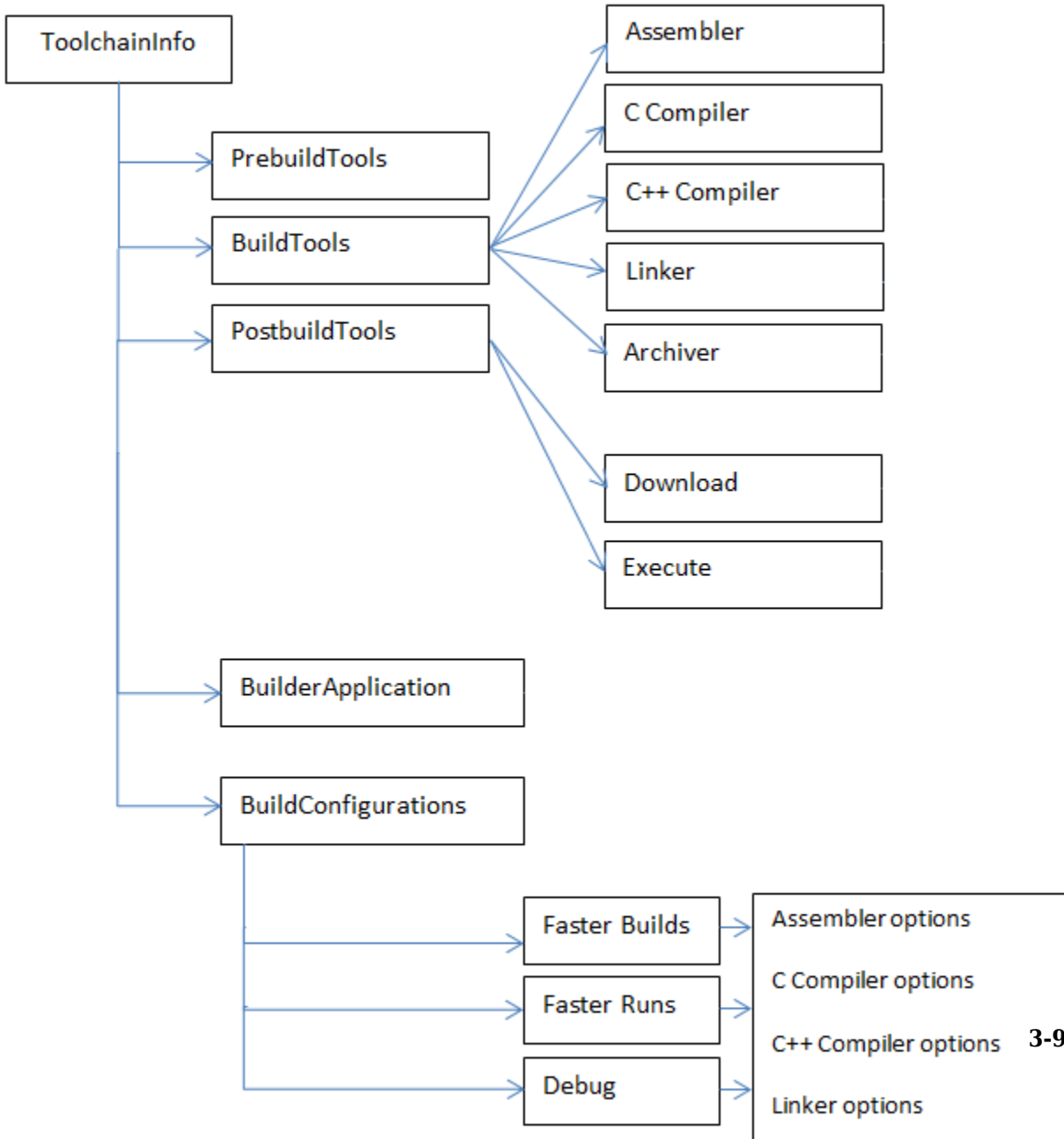
To get toolchain information about defined toolchains, use `getDefaultToolchain` and `getToolchainInfoFromRegistry`.

A `coder.make.ToolchainInfo` object contains:

- `coder.make.BuildTool` objects that can describe each build tool
- `coder.make.BuildConfiguration` objects that can apply sets of options to the build tools

ToolchainInfo class & key properties

Default build tools and options



Construction

`h = coder.make.ToolchainInfo` creates a default ToolchainInfo on page 3-94 object and assigns it to a handle, `h`.

The default `ToolchainInfo` object includes `BuildTool` objects and configurations for C, C++, and gmake:

- The default value of `SupportedLanguages`, `C/C++`, adds `BuildTool` and `BuildConfiguration` objects for C and C++ compilers to `ToolchainInfo`.
- The default value of `BuildArtifact`, `gmake`, adds a `BuildTool` object for gmake to `ToolchainInfo.BuilderApplication`.

You can use the input arguments (name-value pairs) to override these defaults when you create the `ToolchainInfo` object. Each property is optional. Each property requires a corresponding value. This example overrides the `SupportedLanguages` or `BuildArtifact` defaults.

```
h = coder.make.ToolchainInfo('SupportedLanguages',vLanguages,'
BuildArtifact',vArtifact)
```

The default property values for `SupportedLanguages` or `BuildArtifact` can be overridden only during the creation of the toolchain information object. These properties are read-only after object creation.

Input Arguments

`BuildArtifact` — Name of BuildArtifact property

The property name. For more information, see the “Properties” on page 3-98 description for `BuildArtifact`.

`vArtifact` — Value of BuildArtifact property

`gmake` (default) | `gmake makefile` | `nmake` | `nmake makefile`

Values for the `BuildArtifact` property, specified as a character vector.

`Name` — Name of Name property

The property name. For more information, see the “Properties” on page 3-98 description for `Name`.

`vName` — Value of Name property

Unique name for the toolchain definition, specified as a character vector. The default value is empty.

`Platform` — Name of Platform property

The property name. For more information, see the “Properties” on page 3-98 description for `Platform`.

`vPlatform` — Value of Platform property

`win32` | `glnx86` | `win64` | `glnxa64` | `maci64`

The supported platform, specified as a character vector. The default value is the current platform.

`Revision` — Name of Revision property

The property name. For more information, see the “Properties” on page 3-98 description for `Revision`.

`vRevision` — Value of Revision property

`1.0` (default)

Revision number for ToolchainInfo, specified as a character vector.

`SupportedLanguages` — Name of SupportedLanguages property

The property name. For more information, see the “Properties” on page 3-98 description for `SupportedLanguages`.

`vLanguages` — Value of SupportedLanguages property

`C/C++` (default) | `C` | `C++` | `Asm/C` | `Asm/C/C++` | `Asm/C++`

Supported language or languages, specified as a character vector.

`SupportedVersion` — Name of SupportedVersion property

The property name. For more information, see the “Properties” on page 3-98 description for `SupportedVersion`.

`vVersion` — Value of SupportedVersion property

Version of software build tools that ToolchainInfo supports, specified as a character vector. The default value is empty

Output Arguments

h — ToolchainInfo object handle

variable

A coder.make.ToolchainInfo on page 3-94 object, specified using an object handle, such as `h`. To create `h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

Properties

`Attributes` — Custom attributes of toolchain

Custom attributes of the toolchain

Add custom attributes required by the toolchain and specify their default values.

By default, the list of custom attributes is empty.

Attributes returns a `coder.make.util.UnorderedList`.

For example, the `intel_tc.m` file from “Adding a Custom Toolchain”, defines the following custom attributes:

```
tc.addAttribute('TransformPathsWithSpaces');
tc.addAttribute('RequiresCommandFile');
tc.addAttribute('RequiresBatchFile');
```

To display the `Attributes` list from that example in a MATLAB Command Window, enter:

```
h = intel_tc;
h.Attributes


ans =


# ------------------
```

```
# "Attributes" List
# -------------------
RequiresBatchFile        = true
RequiresCommandFile      = true
TransformPathsWithSpaces = true
```

Use the following methods with `Attributes`:

- `addAttribute`
- `getAttribute`
- `getAttributes`
- `isAttribute`
- `removeAttribute`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`BuildArtifact` — Type of makefile or build artifact

The type of makefile (build artifact) MATLAB Coder uses during the software build process.

Initialize this property when you create `coder.make.ToolchainInfo`. Use the default value, `gmake makefile`, or override the default value using a name-value pair argument, as described in “Construction” on page 3-96.

For example:

```
h = coder.make.ToolchainInfo('BuildArtifact','nmake');
```

The values can be:

- `'gmake'` or `'gmake makefile'` — The GNU make utility
- `'nmake'` or `'nmake makefile'` — The Windows make utility

For example, to display the value of `BuildArtifact` in a MATLAB Command Window, enter:

```
h = coder.make.ToolchainInfo;
h.BuildArtifact
```

```
ans  =

gmake makefile
```

`ToolchainInfo` uses the value of the `BuildArtifact` property to create a `BuildTool` object for the build artifact in `coder.make.ToolchainInfo.BuilderApplication`.

The `intel_tc.m` file from the "Adding a Custom Toolchain" example uses the following line to set the value of `BuildArtifact`:

```
tc = coder.make.ToolchainInfo('BuildArtifact','nmake makefile');
```

There are no methods to use with `BuildArtifact`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`BuildConfigurations` — List of build configurations

List of build configurations

Each entry in this list is a `coder.make.BuildConfiguration` object.

For example, the `intel_tc.m` file from "Adding a Custom Toolchain", uses the following lines to define the build configurations:

```
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOffOpts));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOffOpts));
cfg.setOption('Linker',linkerOpts);
cfg.setOption('Shared Library Linker',sharedLinkerOpts);
cfg.setOption('Archiver',archiverOpts);

cfg = tc.getBuildConfiguration('Faster Runs');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOnOpts));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOnOpts));
cfg.setOption('Linker',linkerOpts);
cfg.setOption('Shared Library Linker',sharedLinkerOpts);
cfg.setOption('Archiver',archiverOpts);

cfg = tc.getBuildConfiguration('Debug');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOffOpts,debugFlag.CCompiler));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOffOpts,debugFlag.CppCompiler));
cfg.setOption('Linker',horzcat(linkerOpts,debugFlag.Linker));
cfg.setOption('Shared Library Linker',horzcat(sharedLinkerOpts,debugFlag.Linker));
cfg.setOption('Archiver',horzcat(archiverOpts,debugFlag.Archiver));

tc.setBuildConfigurationOption('all','Download','');
```

```
tc.setBuildConfigurationOption('all','Execute','');
tc.setBuildConfigurationOption('all','Make Tool','-f $(MAKEFILE)');
```

To display the `BuildConfigurations` list from that example in a MATLAB Command Window, enter:

```
h = intel_tc;
h.BuildConfigurations

ans =


# ----------------------------
# "BuildConfigurations" List
# ----------------------------
Debug         = <coder.make.BuildConfiguration>
Faster Builds = <coder.make.BuildConfiguration>
Faster Runs   = <coder.make.BuildConfiguration>
```

Use the following methods with `BuildConfigurations`:

- `getBuildConfiguration`
- `removeBuildConfiguration`
- `setBuildConfiguration`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`BuildTools` — List of build tools in toolchain

The list of build tools in the toolchain.

Each entry in this list is a `coder.make.BuildTool` object.

When you initialize `ToolchainInfo`, the `SupportedLanguages` property determines which build tools are created in `BuildTools`. For more information, see `SupportedLanguages` or “Construction” on page 3-96.

The `BuildTool` objects ToolchainInfo can create based on the `SupportedLanguages` are:

- Assembler
- C Compiler
- C++ Compiler
- Linker
- Archiver

For example, the `intel_tc.m` file from “Adding a Custom Toolchain”, uses the following lines to get and update one of the `BuildTool` objects:

```
% ----------------------------
% C Compiler
% ----------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

To display the `BuildTools` list from that example in a MATLAB Command Window, enter:

```
h = intel_tc;
h.BuildTools
```

```
ans =


# --------------------
# "BuildTools" List
# --------------------
C Compiler   = <coder.make.BuildTool>
C++ Compiler = <coder.make.BuildTool>
Archiver     = <coder.make.BuildTool>
Linker       = <coder.make.BuildTool>
MEX Tool     = <coder.make.BuildTool>
```

Use the following methods with `BuildTools`:

- addBuildTool
- getBuildTool
- removeBuildTool
- setBuildTool

Attributes:

GetAccess	public
SetAccess	public

BuilderApplication — Properties of build tool

Properties of the build tool that runs the makefile or build artifact

ToolchainInfo uses the value of the BuildArtifact property to create a BuildTool object for coder.make.ToolchainInfo.BuilderApplication, as described in “Construction” on page 3-96.

For example, the intel_tc.m file from “Adding a Custom Toolchain”, uses the following lines to set the BuildArtifact and update BuilderApplication objects:

```
h = coder.make.ToolchainInfo('BuildArtifact','nmake');
```

To display the value of BuilderApplication from that example in a MATLAB Command Window, enter:

```
h.BuilderApplication
```

```
ans =

##########################################
# Build Tool: NMAKE Utility
##########################################

Language              : ''
OptionsRegistry       : {'Make Tool','MAKE_FLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {'*'}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<|'

# ---------
# Command
# ---------
MAKE = nmake
MAKE_PATH  =
```

```
# ------------
# Directives
# ------------
Comment                = #
DeleteCommand          = @del
DisplayCommand         = @echo
FileSeparator          = \
ImpliedFirstDependency = $<
ImpliedTarget          = $@
IncludeFile            = !include
LineContinuation       = \
MoveCommand            = @mv
ReferencePattern       = \$\($1\)
RunScriptCommand       = @cmd /C

# -----------------
# File Extensions
# -----------------
Makefile = .mk
```

Use the `setBuilderApplication` method with `BuilderApplication`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`InlinedCommands` — Commands toolchain needs to inline within generated makefile

Commands the toolchain needs to inline within the generated makefile

Specify inlined commands to insert verbatim into the makefile. The default value is `empty`.

The datatype is character vector.

For example, to display and then update the value of the `InlinedCommands` property, use the MATLAB Command Window to enter:

```
h.InlinedCommands


ans  =

     ''

h.InlinedCommands =  '!include <ntwin32.mak>';
h.InlinedCommands
```

```
!include <ntwin32.mak>
```

The “Adding a Custom Toolchain” example does not include the `InlinedCommands` property.

There are no methods to use with `InlinedCommands`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`MATLABCleanup` — MATLAB cleanup commands

MATLAB cleanup commands

Specify MATLAB commands or scripts to perform cleanup routines specific to this toolchain. Use commands or scripts that can be invoked from the MATLAB Command Window. The default value is empty.

The datatype is a cell array of character vectors.

For example, to display and then update the value of the `MATLABSetup` and `MATLABCleanup` properties, use the MATLAB Command Window to enter:

```
h = coder.make.ToolchainInfo;
h.MATLABSetup;
h.MATLABCleanup;
h.MATLABSetup{1}    = sprintf('if ispc \n origTMP=getenv(''TMP''); \n setenv(''TMP'',''C:\\TEMP'');\nend');
h.MATLABCleanup{1}  = sprintf('if ispc \n setenv(''TMP'',origTMP); \nend');
```

The following list illustrates where this property fits in the sequence of operations :

1. MATLAB Setup
2. Shell Setup
3. Prebuild
4. Build (assembler, compilers, linker, archiver)
5. Postbuild
    a. Download
    b. Execute
6. Shell Cleanup

7 MATLAB Cleanup

The “Adding a Custom Toolchain” example does not include the `MATLABCleanup` property.

There are no methods to use with `MATLABCleanup`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`MATLABSetup` — MATLAB setup commands

MATLAB setup commands

Specify MATLAB commands or scripts to perform setup routines specific to this toolchain. Use commands or scripts that can be invoked from the MATLAB Command Window. The default value is empty.

The datatype is a cell array of character vectors.

For example, to display and then update the value of the `MATLABSetup` and `MATLABCleanup` properties, use the MATLAB Command Window to enter:

```
h = coder.make.ToolchainInfo;
h.MATLABSetup;
h.MATLABCleanup;
h.MATLABSetup{1}    = sprintf('if ispc \n origTMP=getenv(''TMP''); \n setenv(''TMP'',''C:\\TEMP'');\nend');
h.MATLABCleanup{1}  = sprintf('if ispc \n setenv(''TMP'',origTMP); \nend');
```

The following list illustrates where this property fits in the sequence of operations :

1. MATLAB Setup
2. Shell Setup
3. Prebuild
4. Build (assembler, compilers, linker, archiver)
5. Postbuild
    a. Download
    b. Execute
6. Shell Cleanup

7 MATLAB Cleanup

The “Adding a Custom Toolchain” example does not include the `MATLABSetup` property.

There are no methods to use with `MATLABCleanup`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Macros` — List of custom macros

List of custom macros that contains macro names and values

The list is a `coder.make.util.OrderedList` of `coder.make.BuildItem` objects.

By default this list is empty. For example:

```
h = coder.make.ToolchainInfo;
h.Macros
```

```
ans =


# ---------------
# "Macros" List
# ---------------
(empty)
```

ToolchainInfo uses macros in two ways:

- It writes macros that are used by the current build to the makefile as variables. For example:

  ```
  TI_INSTALL      =  C:\Program Files\CCSv4
  TI_C2000_TOOLS      =  $(TI_INSTALL)\tools\compiler\c2000\bin
  ```

- When the custom toolchain has been registered, validate expands the complete path provided by a macro, including macros contained within macros. For example, when ToolchainInfo validates the path in the following compiler information, it expands both `TI_C2000_TOOLS` and `TI_INSTALL`:

  ```
  Command = 'cl2000'
  Path = '$(TI_C2000_TOOLS)'
  ```

The default value of `Macros` is an empty list.

The datatype is `coder.make.util.OrderedList` of `coder.make.BuildItem` objects.

For example, the `intel_tc.m` file from "Adding a Custom Toolchain" uses the following lines to add macros to `Macros`:

```
% ------------------------------
% Macros
% ------------------------------
tc.addMacro('MW_EXTERNLIB_DIR',['$(MATLAB_ROOT)\extern\lib\' tc.Platform '\microsoft']);
tc.addMacro('MW_LIB_DIR',['$(MATLAB_ROOT)\lib\' tc.Platform]);
tc.addMacro('CFLAGS_ADDITIONAL','-D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('CPPFLAGS_ADDITIONAL','-EHs -D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('LIBS_TOOLCHAIN','$(conlibs)');
tc.addMacro('CVARSFLAG','');

tc.addIntrinsicMacros({'ldebug','conflags','cflags'});
```

With that example, to see the corresponding property values in a MATLAB command window, enter:

```
h = intel_tc;
h.Macros
```

```
ans =


# ---------------
# "Macros" List
# ---------------
MW_EXTERNLIB_DIR    = $(MATLAB_ROOT)\extern\lib\win64\microsoft
MW_LIB_DIR          = $(MATLAB_ROOT)\lib\win64
CFLAGS_ADDITIONAL   = -D_CRT_SECURE_NO_WARNINGS
CPPFLAGS_ADDITIONAL = -EHs -D_CRT_SECURE_NO_WARNINGS
LIBS_TOOLCHAIN      = $(conlibs)
CVARSFLAG           =
ldebug              =
conflags            =
cflags              =
```

Use the following methods with this property:

- `addMacro`
- `getMacro`
- `removeMacro`

- setMacro
- addIntrinsicMacros
- removeIntrinsicMacros

Attributes:

GetAccess	public
SetAccess	public

Name — Unique name for toolchain definition

Unique name for the toolchain definition

Specify the full name of the toolchain. This name also appears as one of the **Toolchain** parameter options on the **Hardware** tab of the project build settings. The default value is empty. The recommended format is:

name version | *build artifact* (*platform*)

The datatype is character vector.

For example, the `intel_tc.m` file from "Adding a Custom Toolchain" uses the following line to define the value of `Name`:

```
tc.Name             = 'Intel v12.1 | nmake makefile (64-bit Windows)';
```

With that example, to see the corresponding property values in the MATLAB Command Window, enter:

```
h = intel_tc;
h.Name


ans  =

Intel v12.1 | nmake makefile (64-bit Windows)
```

Attributes:

GetAccess	public
SetAccess	public

`Platform` — Specify supported platform

Specify the supported platform

Specify the platform upon which the toolchain will be used. The default value is the current platform. Supported values are `win32`, `win64`, `maci64`, and `glnxa64`.

Create a separate ToolchainInfo for each platform.

The datatype is character vector.

This property does not have any associated methods. Assign the value directly to the `Platform`.

For example, the `intel_tc.m` file from “Adding a Custom Toolchain” uses the following line to define the value of `Platform`:

```
tc.Platform         = 'win64';
```

With that example, to see the corresponding property values in a MATLAB Command Window, enter:

```
h = intel_tc;
h.Platform


ans  =

win64
```

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`PostbuildTools` — List of tools used after linker archiver

The list of tools used after the linker/archiver are invoked.

The list is a `coder.make.util.OrderedList` of `coder.make.BuildTool` objects.

By default the list contains two `BuildTool` objects: `Download` and `Execute`.

To see the corresponding property values in the MATLAB Command Window, enter:

```
h = coder.make.ToolchainInfo;
h.PostbuildTools


ans =


# -----------------------
# "PostbuildTools" List
# -----------------------
Download = <coder.make.BuildTool>
Execute  = <coder.make.BuildTool>
```

The “Adding a Custom Toolchain” example does not include the `PostbuildTools` property.

Use the following methods with this property:

- `addPostbuildTool`
- `getPostbuildTool`
- `removePostbuildTool`
- `setPostbuildTool`

Attributes:

`Download`	`public`
`Execute`	`public`

`PrebuildTools` — List of tools used before compiling source files

The list of tools used before compiling the source files into object files.

The list is a `coder.make.util.OrderedList` of `coder.make.BuildTool` objects.

By default this list is empty. For example:

```
h.PrebuildTools


ans =


# ----------------------
```

```
# "PrebuildTools" List
# --------------------
(empty)
```

The “Adding a Custom Toolchain” example does not include the `PrebuildTools` property.

Use the following methods with this property:

- `addPrebuildTool`
- `getPrebuildTool`
- `removePrebuildTool`
- `setPrebuildTool`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Revision` — Assign revision number to ToolchainInfo

Assign revision number to ToolchainInfo on page 3-94

The author of the toolchain definition file can use this information to differentiate one version of the file from another. The default value is `1.0`.

The datatype is character vector.

This property does not have any associated methods. Assign the value directly to the `Revision`.

For example:

```
h.Revision


ans  =

1.0


h.Revision = '2.0';
h.Revision
```

```
ans  =

2.0
```

Attributes:

GetAccess	public
SetAccess	public

ShellCleanup — Shell scripts that clean up toolchain

Shell scripts that clean up the toolchain

Specify shell commands or scripts to perform cleanup routines specific to this toolchain. Use commands or scripts that can be invoked from the system command environment. The default value is empty.

The datatype is a cell array of character vectors. Each character vector is a shell cleanup command.

If ToolchainInfo invokes a setup routine, you can use a corresponding set of cleanup routines to restore the system environment to its original settings. For example, if a setup routine added environment variables and folders to the system path, you can use a cleanup routine to remove them.

For example:

```
>> h.ShellCleanup

ans  =

     []

>> h.ShellCleanup = 'call "cleanup_mssdk71.bat"';

>> h.ShellCleanup

ans =

    'call "cleanup_mssdk71.bat"'
```

The following list illustrates where this property fits in the sequence of operations :

1. MATLAB Setup
2. Shell Setup
3. Prebuild
4. Build (assembler, compilers, linker, archiver)
5. Postbuild
   a. Download
   b. Execute
6. Shell Cleanup
7. MATLAB Cleanup

The “Adding a Custom Toolchain” example does not include the `ShellCleanup` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ShellSetup` — Shell scripts that set up toolchain

Shell scripts that set up the toolchain

Specify shell commands or scripts to perform setup routines specific to this toolchain. Use commands or scripts that can be invoked from the system command environment. The default value is empty.

The datatype is a cell array of character vectors. Each character vector is a shell setup command.

If ToolchainInfo invokes a setup routine, you can use a corresponding set of cleanup routines to restore the system environment to its original settings. For example, if a setup routine added environment variables and folders to the system path, you can use a cleanup routine to remove them.

For example:

```
>> h.ShellSetup

ans  =
```

```
     []

>> h.ShellSetup = 'call "setup_mssdk71.bat"';

>> h.ShellSetup

ans =

    'call "setup_mssdk71.bat"'
```

The `intel_tc.m` file in “Adding a Custom Toolchain” uses the following lines to set the value of `ShellSetup`:

```
% ------------------------------
% Setup
% ------------------------------
% Below we are using %ICPP_COMPILER12% as root folder where Intel Compiler is
% installed. You can either set an environment variable or give full path to the
% compilervars.bat file
tc.ShellSetup{1} = 'call %ICPP_COMPILER12%\bin\compilervars.bat intel64';
```

With that example, to see the corresponding property values in the MATLAB Command Window, enter:

```
h = intel_tc;
h.ShellSetup


ans =

    'call %ICPP_COMPILER12%\bin\compilervars.bat intel64'
```

The following list illustrates where this property fits in the sequence of operations :

1. MATLAB Setup
2. Shell Setup
3. Prebuild
4. Build (assembler, compilers, linker, archiver)
5. Postbuild
    a. Download
    b. Execute

6 Shell Cleanup

7 MATLAB Cleanup

Attributes:

GetAccess	public
SetAccess	public

SupportedLanguages — Create BuildTool objects for specific languages

Create `BuildTool` objects for specific languages

Initializing `ToolchainInfo` creates `BuildTool` objects for the language or set of languages specified by SupportedLanguages on page 3-116.

If you do not specify a value for SupportedLanguages on page 3-116, the default value is `'C/C++'`. This adds `BuildTool` objects for a C compiler and a C++ compiler to the other `BuildTool` objects in ToolchainInfo on page 3-94.

To override the default, use a name-value pair to specify a value for SupportedLanguages on page 3-116 when you initialize ToolchainInfo on page 3-94. For example:

```
h = coder.make.ToolchainInfo('SupportedLanguages','C');
```

The value can be: `'C'`, `'C++'`, `'C/C++'`, `'Asm/C'`, `'Asm/C++'`, or `'Asm/C/C++'`.

The SupportedLanguages on page 3-116 property does not have any related methods.

The “Adding a Custom Toolchain” example does not include the SupportedLanguages on page 3-116 property.

Attributes:

GetAccess	public
SetAccess	protected

SupportedVersion — Version of software build tools that ToolchainInfo supports

The version of the software build tools `ToolchainInfo` supports.

The default value is empty.

The datatype is character vector.

This property does not have any associated methods. Assign the value directly to the `SupportedVersion`.

With the “Adding a Custom Toolchain” example, the value of `SupportedVersion` is defined in the `intel_tc.m` toolchain definition file:

```
tc.SupportedVersion = '12.1';
```

With that example, to see the corresponding property values in the MATLAB command window, enter:

```
h = intel_tc;
h.SupportedVersion


ans  =

12.1
```

Attributes:

GetAccess	public
SetAccess	public

Methods

addAttribute	Add custom attribute to `Attributes`
addBuildConfiguration	Add build configuration
addBuildTool	Add `BuildTool` object to `BuildTools`
addIntrinsicMacros	Add intrinsic macro to `Macros`
addMacro	Add macro to `Macros`
addPostbuildTool	Add postbuild tool to `PostbuildTools`
addPostDownloadTool	Add post-download tool to `PostDownloadTool`
addPostExecuteTool	Add post-execute tool to `PostbuildTools`
addPrebuildTool	Add prebuild tool to `PrebuildTools`
cleanup	Run cleanup commands
getAttribute	Get value of attribute
getAttributes	Get list of attribute names
getBuildConfiguration	Get handle for build configuration object
getBuildTool	Get `BuildTool` object
getMacro	Get value of macro
getPostbuildTool	Get postbuild `BuildTool` object
getPrebuildTool	Get prebuild `BuildTool` object
getSupportedLanguages	Get list of supported languages
isAttribute	Determine if attribute exists
removeAttribute	Remove attribute
removeBuildConfiguration	Remove build configuration
removeBuildTool	Remove `BuildTool` object from `BuildTools`
removeIntrinsicMacros	Remove intrinsic macro
removeMacro	Remove macro from `Macros`
removePostbuildTool	Remove postbuild build tool
removePrebuildTool	Remove prebuild build tool
setBuildConfiguration	Set value of specified build configuration
setBuildConfigurationOption	Sets value of build tool options for build configuration
setBuilderApplication	Update builder application to work on specific platform
setBuildTool	Assign `BuildTool` object to named build tool in `BuildTools`
setMacro	Set value of macro
setPostbuildTool	Assign `BuildTool` object to `PostbuildTool` tool in `PostbuildTools`

See Also

`getDefaultToolchain` | `getToolchainInfoFromRegistry` | `coder.make.BuildConfiguration` | `coder.make.BuildItem` | `coder.make.BuildTool`

Topics

"Adding a Custom Toolchain"
"About coder.make.ToolchainInfo"

target Package

Register new target hardware

Description

Manage target hardware information

Classes

target.LanguageImplementation	Provide C and C++ compiler implementation details
target.Processor	Provide target processor information

Functions

target.add	Add target feature object to MATLAB memory
target.create	Create target feature object
target.get	Retrieve target feature object from MATLAB memory
target.remove	Remove target feature object from MATLAB memory

See Also

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.add

Package: target

Add target feature object to MATLAB memory

Syntax

```
target.add(targetFeatureObject)
target.add(targetFeatureObject, 'UserInstall', dataPersistence)
```

Description

`target.add(targetFeatureObject)` adds the specified target feature object to MATLAB memory. By default, the target data is available only for the current MATLAB session.

`target.add(targetFeatureObject, 'UserInstall', dataPersistence)` controls persistence of target data over MATLAB sessions.

Examples

Create New Hardware Implementation

For workflow examples that use this function, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

Input Arguments

`targetFeatureObject` — Target feature object
object

Specify the target feature object that you want to add to MATLAB memory.

Example: `target.add(myTargetFeatureObject);`

`dataPersistence` — Target data persistence
false (default) | true

Control persistence of target data in MATLAB memory:

- `true` –- Target data persists over multiple MATLAB sessions.
- `false` –- Target data is available only for the current MATLAB session.

Example: `target.add(myTargetFeatureObject, 'UserInstall', true);`

Data Types: `logical`

See Also

`target.create` | `target.get` | `target.remove`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.create

Package: target

Create target feature object

Syntax

```
targetFeatureObject = target.create(targetFeatureClass)
targetFeatureObject = target.create(targetFeatureClass,Name,Value)
```

Description

`targetFeatureObject = target.create(targetFeatureClass)` creates and returns an object of the specified class.

`targetFeatureObject = target.create(targetFeatureClass,Name,Value)` configures the object using one or more name-value pair arguments.

Examples

Create New Hardware Implementation

For workflow examples that use this function, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

Input Arguments

`targetFeatureClass` — Target feature class
character vector | string

Specify class of object:

- `'Processor'`-- Create `target.Processor` object.
- `'LanguageImplementation'`-- Create `target.LanguageImplementation` object.
- `'Alias'`-- Create `target.Alias` object.

Example: `'Processor'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `myLangImp = target.create('LanguageImplementation', 'Name', 'myLanguageImplementation', 'Copy', 'ARM Compatible-ARM Cortex');`

Example: `myProc = target.create('Processor', 'Name', 'myProcessor', 'Manufacturer', 'myProcessorManufacturer');`

`Copy` — Copy existing target feature object
character vector | string

Create a target feature object by copying values from an existing target feature object.

***`propertyName`* — Property name**
character vector | string

Create the target feature object with properties that are set to values that you specify.

Output Arguments

`targetFeatureObject` — Target feature object
object

The returned object is a:

- `target.Processor` object if `targetFeatureClass` is `'Processor'`
- `target.LanguageImplementation` object if `targetFeatureClass` is `'LanguageImplementation'`
- `target.Alias` object if `targetFeatureClass` is `'Alias'`

See Also

`target.add` | `target.get` | `target.remove`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.get

Package: target

Retrieve target feature object from MATLAB memory

Syntax

```
targetFeatureObject = target.get(targetFeatureClass,
targetFeatureObjectId)
```

Description

`targetFeatureObject = target.get(targetFeatureClass, targetFeatureObjectId)` retrieves a target feature object from MATLAB memory.

Examples

Remove Target Feature Object

This example shows how you can remove a `target.LanguageImplementation` object associated with an object identifier, *`myLanguageImplementationID`*.

Retrieve the object from MATLAB memory.

```
objectToRemove = target.get('LanguageImplementation', myLanguageImplementationID);
```

Remove the object.

```
target.remove(objectToRemove);
```

Input Arguments

`targetFeatureClass` — Target feature class
character vector | string

Specify the class of the object that you want to retrieve. For example, to retrieve:

- A `target.Processor` object, specify `'Processor'` .
- A `target.LanguageImplementation` object, specify `'LanguageImplementation'`.

`targetFeatureObjectId` — Target feature object identifier
character vector | string

Specify the unique identifier of the object that you want to retrieve, that is, the `Id` property value of the object.

Output Arguments

`targetFeatureObject` — Target feature object
object

Retrieved target feature object. For example:

- If `targetFeatureClass` is `'Processor'`, the returned object is a `target.Processor` object.
- If `targetFeatureClass` is `'LanguageImplementation'`, the returned object is a `target.LanguageImplementation` object.

See Also
`target.add` | `target.create` | `target.remove`

Topics
“Register New Hardware Devices”

Introduced in R2019a

target.LanguageImplementation class

Package: target

Provide C and C++ compiler implementation details

Description

Use the `target.LanguageImplementation` class to provide implementation details about the C and C++ compiler for your target hardware. For example, byte ordering.

To create a `target.LanguageImplementation` object, use the `target.create` function.

Properties

`AtomicFloatSize` — Largest atomic float size

integer

Size in bits of the largest floating-point data type that you can atomically load and store on the hardware

Attributes:

GetAccess

public

SetAccess

public

Data Types: `int32`

`AtomicIntegerSize` — Largest atomic integer size

integer

Size in bits of the largest integer that you can atomically load and store on the hardware

Attributes:

GetAccess
public
SetAccess
public

Data Types: `int32`

`Endianess` — Byte ordering

`'Little'` (default) | `'Big'` | `'Unspecified'`

Byte ordering implemented by target hardware.

Attributes:

GetAccess
public
SetAccess
public

`DataTypes` — Data type definitions

object

`target.DataTypes` object that provides C and C++ data type definitions through properties described in this table.

Property Name	Purpose
`Char`	`target.datatype.Char` object for `char`.
`Short`	`target.datatype.Short` object for `short`.
`Int`	`target.datatype.Int` object for `int`.
`Long`	`target.datatype.Long` object for `long`.
`LongLong`	`target.datatype.LongLong` object for `longlong`.
`Half`	`target.datatype.Half` object for a half precision data type that the target uses.
`Float`	`target.datatype.Float` object for `float`.
`Double`	`target.datatype.Double` object for `double`.

Property Name	Purpose
`Pointer`	`target.datattype.Pointer` object for `pointer`.
`SizeT`	`target.datatype.SizeT` object for `size_t`.
`PtrDiffT`	`target.datatype.PtrDiffT` object for `ptrdiff_t`.

Attributes:

GetAccess
: public

SetAccess
: private

`Id` — Object identifier

character vector | string

Value of `Name` property.

Attributes:

GetAccess
: public

SetAccess
: private

`Name` — Name

character vector | string

Name of the target language implementation

Attributes:

GetAccess
: public

SetAccess
: public

`WordSize` — Native word size

integer

Native word size for the target hardware.

Attributes:

GetAccess
: public

SetAccess
: public

Data Types: `int32`

Examples

Create New Hardware Implementation

For examples that use this class, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

See Also

`target.Processor` | `target.create`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.Processor class

Package: target

Provide target processor information

Description

Use the `target.Processor` class to provide information about your target processor. For example, name, manufacturer, and language implementation.

To create a `target.Processor` object, use the `target.create` function.

Properties

`Id` — Object identifier

character vector | string

The object identifier is the hyphenated combination of the `Manufacturer` and `Name` property values. If the `Manufacturer` property is empty, the object identifier is the `Name` property value.

Attributes:

GetAccess

public

SetAccess

private

`LanguageImplementations` — Language implementation

object

Associated `target.LanguageImplementation` object.

Attributes:

GetAccess
: public

SetAccess
: public

`Name` — Processor name

character vector | string

Name of the target processor.

Example: `'Cortex-A53'`

Attributes:

GetAccess
: public

SetAccess
: public

`Manufacturer` — Processor manufacturer

character vector | string

Optional description of the target processor manufacturer.

Example: `'ARM Compatible'`

Attributes:

GetAccess
: public

SetAccess
: public

Examples

Create New Hardware Implementation

For examples that use this class, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

See Also

`target.LanguageImplementation` | `target.create`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.remove

Package: target

Remove target feature object from MATLAB memory

Syntax

```
target.remove(targetFeatureObject)
target.remove(targetFeatureClass, targetFeatureObjectId)
```

Description

`target.remove(targetFeatureObject)` removes the target feature object from MATLAB memory.

`target.remove(targetFeatureClass, targetFeatureObjectId)` removes the target feature object specified by class and identifier.

Examples

Remove Target Feature Object From Memory

You can specify and save a hardware device implementation to MATLAB memory.

```
armv8 = target.create('LanguageImplementation', ...
                      'Name', 'Armv8-A LP64');
a53 = target.create('Processor', 'Name', 'Cortex-A53', ...
                    'Manufacturer', 'ARM Compatible');
a53.LanguageImplementations = armv8;
target.add(a53)
```

When target feature objects are not required, you can use the function to remove the objects from MATLAB memory.

To remove only the `target.Processor` object, run:

```
target.remove(a53)
```

or:

```
target.remove('Processor', 'ARM Compatible-Cortex-A53');
```

Input Arguments

`targetFeatureObject` — Target feature object

object

Specify the target feature object that you want to remove.

`targetFeatureClass` — Target feature class

character vector | string

Specify the class of the target feature object that you want to remove. For example:

- If the class is `target.Processor`, specify `'Processor'`.
- If the class is `target.LanguageImplementation`, specify `'LanguageImplementation'`.

Example: `'Processor'`

`targetFeatureObjectId` — Target feature object identifier

character vector | string

Specify the unique identifier of the object that you want to remove, that is, the `Id` property value of the object.

See Also

`target.add` | `target.create` | `target.get`

Topics

“Register New Hardware Devices”

Introduced in R2019a

4

Using Objects Reference

coder.CodeConfig

Configuration parameters for C/C++ code generation from MATLAB code

Description

A `coder.CodeConfig` object contains the configuration parameters that `codegen` uses for generating a static library, a dynamically linked library, or an executable program. Pass the object to the `codegen` function by using the `-config` option.

Creation

Syntax

```
cfg = coder.config(build_type)
cfg = coder.config(build_type, 'ecoder', false)
```

Description

`cfg = coder.config(build_type)` creates a code generation configuration object for the specified build type, which can be a static library, a dynamically linked library, or an executable program. If the Embedded Coder product is not installed, it creates a `coder.CodeConfig` object. Otherwise, it creates a `coder.EmbeddedCodeConfig` object.

`cfg = coder.config(build_type, 'ecoder', false)` creates a `coder.CodeConfig` object for the specified output type even if you have Embedded Coder installed.

Input Arguments

`build_type` — Output to build from generated C/C++ code
`'LIB'` | `'DLL'` | `'EXE'`

Output to build from generated C/C++ code, specified as one of the values in this table.

Value	Description
`'LIB'`	Static library
`'DLL'`	Dynamically linked library
`'EXE'`	Executable program

Properties

`BuildConfiguration` — Compiler optimization or debug settings for toolchain
`'Faster Builds'` (default) | `'Faster Runs'` | `'Debug'` | `'Specify'`

Compiler optimization or debug settings for toolchain, specified as one of the values in this table.

Value	Description
`'Faster Builds'`	Optimizes the build for shorter build times.
`'Faster Runs'`	Optimizes the build for faster running executables.
`'Debug'`	Optimizes the build for debugging.
`'Specify'`	Enables the `CustomToolchainOptions` property for customization of settings for tools in the selected toolchain. If the `Toolchain` property is set to `'Automatically locate an installed toolchain'`, then setting `BuildConfiguration` to `'Specify'` changes `Toolchain` to the located toolchain.

`CodeReplacementLibrary` — Code replacement library for generated code
character vector

Code replacement library for generated code, specified as one of the values in this table:

Value	Description
`'None'`	This value is the default value. Does not use a code replacement library.
Named code replacement library	Generates calls to a specific platform, compiler, or standards code replacement library. The list of named libraries depends on: • Installed support packages. • System target file, language, standard math library, and device vendor configuration. • Whether you created and registered code replacement libraries, using the Embedded Coder product.

Compatible libraries depend on these parameters:

- `TargetLang`
- `TargetLangStandard`
- `ProdHWDeviceType` in the hardware implementation configuration object.

Embedded Coder offers more libraries and the ability to create and use custom code replacement libraries.

MATLAB Coder generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Note MATLAB Coder software does not support TLC callbacks.

`CompileTimeRecursionLimit` — Maximum number of function specializations for compile-time recursion

50 (default) | positive integer

Maximum number of function specializations for compile-time recursion, specified as a positive integer. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to 0. The default compile-time recursion limit is large enough for most recursive functions that require this type of recursion. If code generation fails because of the compile-time recursion limit, and you want compile-time recursion, try to increase the limit. Alternatively, change your MATLAB code so that the code generator uses run-time recursion. See “Compile-Time Recursion Limit Reached”.

`ConstantFoldingTimeout` — Maximum number of instructions to be executed by the constant folder

40000 (default) | positive integer

Maximum number of instructions that the constant folder executes. In some situations, code generation requires specific instructions to be constant. If constant folding stops before these instructions are constant-folded, code generation fails. In this case, increase the value of `ConstantFoldingTimeout`.

See “MATLAB Coder Optimizations in Generated Code”.

`CustomBLASCallback` — BLAS callback class

' ' (default) | character vector

Callback class for BLAS library calls in code generated for certain low-level vector and matrix operations in MATLAB code, specified as a character vector.

If you specify a BLAS callback class, for certain low-level vector and matrix functions, the code generator produces BLAS calls by using the CBLAS C interface to your BLAS library. The callback class provides the name of your CBLAS header file, the names of CBLAS data types, and the information required to link to your BLAS library. If this parameter is empty, the code generator produces code for matrix functions instead of a BLAS call.

See “Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”.

`CustomFFTCallback` — Callback class for FFTW library calls

' ' (default) | character vector

Callback class for FFTW library calls in code generated for FFT functions in MATLAB code, specified as a character vector.

To improve the execution speed of FFT functions, the code generator produces calls to the FFTW library that you specify in the callback class. If this parameter is empty, the

code generator uses its own algorithms for FFT functions instead of calling the FFTW library.

See “Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”.

`CustomHeaderCode` — Custom code that appears at top of generated C/C++ header files

' ' (default) | character vector

Custom code that appears near the top of each C/C++ header file generated from your MATLAB code, specified as a character vector.

`CustomInclude` — Include folders to add to include path for compiling generated code

' ' (default) | character vector

Include folders to add to the include path when compiling the generated code. Specify the list of include folders as a character vector. In the character vector, separate include folders by a `pathsep` character. For example:

```
cfg = coder.config('lib','ecoder',false);
cfg.CustomInclude = ['C:\Project' pathsep 'C:\Custom Files'];
```

`CustomInitializer` — Custom code to include in the generated initialize function

' ' (default) | character vector

Custom code to include in the generated initialize function, specified as a character vector.

`CustomLAPACKCallback` — LAPACK callback class

' ' (default) | character vector

Callback class for LAPACK library calls in code generated for certain linear algebra functions in MATLAB code, specified as a character vector.

If you specify a LAPACK callback class, for certain linear algebra functions, the code generator produces LAPACK calls by using the LAPACKE C interface to your LAPACK library. The callback class provides the name of your LAPACKE header file and the information required to link to your LAPACK library. If this parameter is empty, the code generator produces code for linear algebra functions instead of a LAPACK call.

See “Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”.

`CustomLibrary` — Static library files to link with the generated code
' ' (default) | character vector

Static library files to link with the generated code, specified as a character vector. In the character vector, separate library file names by a `pathsep` character.

`CustomSource` — Source files to compile and link with the generated code
' ' (default) | character vector

Source files to compile and link with the generated code, specified as a character vector. In the character vector, separate source file names by a `pathsep` character.

The build process searches for the source files first in the current folder, and then in the include folders that you specify in `CustomInclude`. If source files with the same name occur in multiple folders on the search path, the build process might use a different file than the file that you specified.

Suppose that you specify `foo.cpp` as a source file. If `foo.c` and `foo.cpp` are both on the search path, you cannot be sure whether the build process uses `foo.c` or `foo.cpp`.

`CustomSourceCode` — Code to appear near the top of the generated `.c` or `.cpp` file
' ' (default) | character vector

Specify code to appear near the top of the generated `.c` or `.cpp` file, outside of any function. Specify code as a character vector.

Do not specify a C static function definition.

`CustomTerminator` — Code that appears in the generated terminate function
' ' (default) | character vector

Code that appears in the generated terminate function, specified as a character vector.

`CustomToolchainOptions` — Custom settings for tools in selected toolchain
cell array

Custom settings for tools in selected toolchain, specified as a cell array.

Dependencies:

- The `Toolchain` property determines which tools and options appear in the cell array.
- Setting the `BuildConfiguration` property to `Specify` enables `CustomToolchainOptions`.

First, get the current settings. For example:

```
cfg = coder.config('lib');
cfg.BuildConfiguration='Specify';
opt = cfg.CustomToolchainOptions
```

Then, edit the values in `opt`.

These values derive from the toolchain definition file and the third-party compiler options. See “Custom Toolchain Registration”.

`DataTypeReplacement` — Data type replacement in generated code
`'CBuiltIn'` | `'CoderTypeDefs'`

Data type replacement in generated code, specified as one of the values in this table.

Value	Description
`'CBuiltIn'`	This value is the default value. The code generator uses built-in C data types.
`'CoderTypeDefs'`	The code generator uses predefined data types from `rtwtypes.h`

`DeepLearningConfig` — Configuration object for deep learning code generation
`coder.MklDNNConfig` object | `coder.ARMNEONConfig` object

Configuration object for code generation for deep learning networks, specified as a `coder.MklDNNConfig` object or a `coder.ARMNEONConfig` object.

A `coder.MklDNNConfig` object contains parameters specific to C++ code generation for deep learning using Intel MKL-DNN. To create a `coder.MklDNNConfig` object, use `coder.DeepLearningConfig`. For example:

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

A `coder.ARMNEONConfig` object contains parameters specific to C++ code generation for deep learning using the ARM Compute Library. To create a `coder.ARMNEONConfig` object, use `coder.DeepLearningConfig`. For example:

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('arm-compute');
```

Code generation for deep learning networks requires Deep Learning Toolbox and the MATLAB Coder Interface for Deep Learning Libraries support package.

See “Code Generation for Deep Learning Networks with MKL-DNN” and “Code Generation for Deep Learning Networks with ARM Compute Library”.

Dependency: If `DeepLearningConfig` is set, `codegen` sets `TargetLang` to `C++`.

`Description` — Object description
`'class CodeConfig: C code generation configuration.'` (default) | character vector

Object description, specified as a character vector.

`DynamicMemoryAllocation` — Dynamic memory allocation mode
`'Threshold'` (default) | `'AllVariableSizeArrays'` | `'Off'`

Dynamic memory allocation mode, specified as one of the values in this table.

Value	Description
`'Threshold'`	This value is the default value. The code generator allocates memory dynamically on the heap for variable-size arrays whose size (in bytes) is greater than or equal to `DynamicMemoryAllocationThreshold`.
`'AllVariableSizeArrays'`	The code generator dynamically allocates memory for all variable-size arrays on the heap.

Value	Description
`'Off'`	The code generator statically allocates memory for variable-size arrays on the stack.

Unbounded variable-size arrays require dynamic memory allocation.

Dependencies:

- `EnableVariableSizing` enables this parameter.
- Setting this `DynamicMemoryAllocation` to `'Threshold'` enables the `DynamicMemoryAllocationThreshold` parameter.

See “Generate Code for Variable-Size Data”.

`DynamicMemoryAllocationThreshold` — Size threshold for dynamic memory allocation of variable-size arrays
`65536` (default) | positive integer

Size threshold for dynamic memory allocation of variable-size arrays, specified as a positive integer. The code generator uses dynamic memory allocation for variable-size arrays whose size (in bytes) is greater than or equal to the threshold.

Dependency:

- Setting `DynamicMemoryAllocation` to `'Threshold'` enables this parameter.

See “Generate Code for Variable-Size Data”.

`EnableAutoExtrinsicCalls` — Automatic extrinsic function calls
`true` (default) | `false`

Automatic extrinsic function calls, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator treats some common visualization functions as extrinsic functions. You do not have to declare these functions as extrinsic by using `coder.extrinsic`. This capability reduces the amount of time that you spend making your code suitable for code generation.
`false`	The code generator does not treat common visualization functions as extrinsic functions unless you declare them as extrinsic by using `coder.extrinsic`.

Some common visualization functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions”.

`EnableMemcpy` — `memcpy` optimization
`true` (default) | `false`

`memcpy` optimization, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the `memcpy` optimization. To optimize code that copies consecutive array elements, the `memcpy` optimization replaces the code with a `memcpy` call. When the number of elements to copy is known at compile time, the code generator uses the `MemcpyThreshold` property to determine whether to use the optimization. See “memcpy Optimization”.
`false`	The code generator does not use the `memcpy` optimization.

`EnableOpenMP` — Parallelization of `parfor`-loops
`true` (default) | `false`

Parallelization of `parfor`-loops, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the OpenMP library to produce loop iterations that run in parallel.
`false`	The code generator treats `parfor`-loops as `for`-loops.

See `parfor`.

Use of the OpenMP library is not compatible with just-in-time (JIT) compilation. If `EnableJIT` and `EnableOpenMP` are `true`, the code generator uses JIT compilation and treats `parfor`-loops as `for`-loops.

`EnableRuntimeRecursion` — Run-time recursion support
`true` (default) | `false`

Run-time recursion support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Recursive functions are allowed in the generated code.
`false`	Recursive functions are not allowed in the generated code.

Some coding standards, such as MISRA®, do not allow recursion. To increase the likelihood of generating code that is compliant with MISRA C®, set `EnableRuntimeRecursion` to `false`.

If your MATLAB code requires run-time recursion and `EnableRuntimeRecursion` is `false`, code generation fails.

See “Code Generation for Recursive Functions”.

`EnableVariableSizing` — Variable-size array support
`true` (default) | `false`

Variable-size array support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Variable-size arrays are allowed for code generation.
`false`	Variable-size arrays are not allowed for code generation.

Dependency:

- Enables `Dynamic memory allocation`.

See “Code Generation for Variable-Size Arrays”.

`FilePartitionMethod` — File partitioning mode
`'MapMFileToCFile'` (default) | `'SingleFile'`

File partitioning mode specified as one of the values in this table.

Value	Description
`'MapMFileToCFile'`	This value is the default value. The code generator produces separate C/C++ files for each MATLAB language file.
`'SingleFile'`	The code generator produces a single file for C/C++ functions that map to your MATLAB entry-point functions. The code generator produces separate C/C++ files for utility functions.

See “How MATLAB Coder Partitions Generated Code”.

`GenCodeOnly` — Generation of only source code
`false` (default) | `true`

Generation of only source code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator produces C/C++ source code and builds object code.
`true`	The code generator produces C/C++ source code, but does not invoke the make command or build object code. When you iterate between modifying MATLAB code and generating C/C++ code, generating only code can save time.

`GenerateComments` — Comments in generated code
`true` (default) | `false`

Comments in generated code, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator places comments in the generated code.
`false`	The code generator does not place comments in the generated code.

`GenerateExampleMain` — Example C/C++ main file generation
`'GenerateCodeOnly'` (default) | `'DoNotGenerate'` | `'GenerateCodeAndCompile'`

Example C/C++ main file generation, specified as one of the values in this table.

Value	Description
`'GenerateCodeOnly'`	This value is the default value. The code generator generates an example C/C++ main function but does not compile it.
`'DoNotGenerate'`	The code generator does not generate an example C/C++ main function.

Value	Description
`'GenerateCodeAndCompile'`	The code generator generates an example C/C++ main function and compiles it to create a test executable. This executable does not return output. If the `GenCodeOnly` parameter is `true`, the code generator does not compile the C/C++ main function.

An example main function is a template to help you to write a C/C++ main function that calls generated C/C++ code. See “Incorporate Generated Code Using an Example Main Function”.

`GenerateMakefile` — Makefile generation
`true` (default) | `false`

Makefile generation during the build process, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator generates a makefile during the build process.
`false`	The code generator does not generate a makefile during the build process. Specify instructions for post-code-generation processing, including compilation and linking, in a post-code-generation command. See “Build Process Customization”.

`GenerateNonFiniteFilesIfUsed` — Generate support files for nonfinite data only if nonfinite data is used
`true` (default) | `false`

Generation of support files for nonfinite data, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator produces the support files for nonfinite data (`Inf` and `NaN`) only if the generated code uses nonfinite data.
`false`	The code generator always produces the support files for nonfinite data (`Inf` and `NaN`).

Dependency:

- Setting `SupportNonFinite` to `true` enables this parameter.

`GenerateReport` — Code generation report
`false` (default) | true

Code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator produces a report only if error or warning messages occur, or if you set `LaunchReport` to `true`.
`true`	The code generator produces a code generation report.

`Hardware` — Object that specifies a hardware board
`coder.Hardware` object

Object that specifies a hardware board. To create the `coder.Hardware` object, use `coder.hardware`. For example:

```
cfg = coder.config('lib');
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Before you use `coder.hardware`, you must install the support package for the hardware.

Dependencies:

- Setting `Hardware` customizes the hardware implementation object and other configuration parameters for a particular hardware board.
- If `DeepLearningConfig` is set to a `coder.ARMNEONConfig` object and `Hardware` is empty, then `codegen` sets the `GenCodeOnly` property to `true`.

Note:

- Suppose that you create a `coder.CodeConfig` object `cfg` in a MATLAB session and use it in another MATLAB session. If the MATLAB host computer for the second session does not have the hardware board specified in the `cfg.Hardware` property installed on it, this parameter reverts to its default value. The default value is `[]`.

`HardwareImplementation` — Hardware implementation object
`coder.HardwareImplementation` object.

Hardware implementation object that specifies hardware-specific configuration parameters for C/C++ code generation. `coder.config` creates a `coder.CodeConfig` object with the `HardwareImplementation` property set to a `coder.HardwareImplementation` object with default parameter values for the MATLAB host computer.

`HighlightPotentialRowMajorIssues` — Potential row-major layout issues
`true` (default) | `false`

Display of potential row-major layout efficiency issues, specified as one of the values in this table.

Value	Description
`true`	The code generation report displays potential efficiency issues due to row-major layout. (This value is the default value.)
`false`	The code generation report does not display issues related to array layout.

See “Code Design for Row-Major Array Layout”.

`InitFltsAndDblsToZero` — Assignment of float and double zero with memset
`true` (default) | `false`

Assignment of float and double zero with `memset`, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the `memset` optimization for assignment of floating-point zero to consecutive array elements. To assign consecutive array elements, the `memset` optimization uses a `memset` call. When the number of elements to assign is known at compile time, the code generator uses the `MemcpyThreshold` property to determine whether to use the optimization. See “memset Optimization”.
`false`	The code generator does not use the `memset` optimization for assignment of float and double zero to consecutive array elements.

`InlineStackLimit` — Stack size limit for inlined functions
4000 (default) | positive integer

Stack size limit for inlined functions, specified as a positive integer. The stack size limit determines the amount of stack space allocated for local variables of the inlined function.

Specifying a limit for the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even when the function returns.

This capability is especially important for embedded processors where stack size can be limited.

See “Control Inlining”.

`InlineThreshold` — Function size threshold for inlining
10 (default) | positive integer

Function size threshold for inlining, specified as a positive integer.

Unless other conditions prevent inlining, the code generator inlines functions that are smaller than the threshold.

The function size is measured as an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. To obtain the inlining behavior that you want, experiment with the threshold. For example, if the default threshold results in inlining of large functions and generation of large amounts of C code, tune the threshold until you are satisfied with the size of the generated code.

See “Control Inlining”.

`InlineThresholdMax` — Maximum size of functions after inlining
200 (default) | positive integer

Maximum size of functions after inlining, specified as a positive integer. If, after inlining, the size of the calling function exceeds `InlineThresholdMax`, then the code generator does not inline the called function.

The function size is measured as an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. To obtain the inlining behavior that you want, experiment with the threshold. For example, if the default threshold results in inlining of large functions and generation of large amounts of C code, tune the threshold until you are satisfied with the size of the generated code.

See “Control Inlining”.

`LaunchReport` — Automatic open of code generation report
`false` (default) | true

Automatic open of code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. If errors or warnings occur, or if `GenerateReport` is `true`, the code generator produces a report, but does not open the report.
`true`	The code generator produces and opens a code generation report.

`LoopUnrollThreshold` — Threshold on the number of iterations that determines whether to automatically unroll a `for`-loop
5 (default) | positive integer

Loops with fewer iterations than this threshold are candidates for automatic unrolling by the code generator. This threshold applies to all `for`-loops in your MATLAB code. For an individual `for`-loop, a `coder.unroll` directive placed immediately before the loop takes precedence over the loop unrolling optimization. The threshold can also apply to some `for`-loops produced during code generation.

See “Unroll for-Loops”.

`MATLABSourceComments` — Inclusion of MATLAB source code as comments in generated code
`false` (default) | `true`

Inclusion of MATLAB source code as comments in generated code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not insert MATLAB source code as comments in the generated code. The code generator does not include the MATLAB function signature in the function banner.

Value	Description
`true`	The code generator inserts MATLAB source code as comments in the generated code. A traceability tag immediately precedes each line of source code. The traceability tag helps you to locate the corresponding MATLAB source code. See “Tracing Generated C/C++ Code to MATLAB Source Code”. If you have Embedded Coder, in the code generation report, the traceability tag links to the source code. The code generator also includes the MATLAB function signature in the function banner.

Dependency:

- `GenerateComments` enables this parameter.

See “Tracing Generated C/C++ Code to MATLAB Source Code”.

`MaxIdLength` — Maximum number of characters in generated identifiers
31 (default) | positive integer

Maximum number of characters in generated identifiers, specified as a positive integer in the range [31, 256]. This property applies to generated function, type definition, and variable names. To avoid truncation of identifiers by the target C compiler, specify a value that matches the maximum identifier length of the target C compiler.

This property does not apply to exported identifiers, such as the generated names for entry-point functions or emxArray API functions. If the length of an exported identifier exceeds the maximum identifier length of the target C compiler, the target C compiler truncates the exported identifier.

`MemcpyThreshold` — Minimum size for `memcpy` or `memset` optimization
64 (default) | positive integer

Minimum size, in bytes, for memcpy or memset optimization, specified as a positive integer.

To optimize generated code that copies consecutive array elements, the code generator tries to replace the code with a `memcpy` call. To optimize generated code that assigns a literal constant to consecutive array elements, the code generator tries to replace the code with a `memset` call.

The number of bytes is the number of array elements to copy or assign multiplied by the number of bytes required for the C/C++ data type.

If the number of elements to copy or assign is variable (not known at compile time), the code generator ignores the `MemcpyThreshold` property.

See “memcpy Optimization” and “memset Optimization”.

`MultiInstanceCode` — Multi-instance, reentrant code
`false` (default) | `true`

Multi-instance, reentrant code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not produce multi-instance, reentrant code.
`true`	The code generator produces reusable, multi-instance code that is reentrant.

See “Reentrant Code”.

`Name` — Object name
`'MexCodeConfig'` (default) | character vector

Object name, specified as a character vector.

`OutputType` — Output to build from generated C/C++ code
`'LIB'` | `'DLL'` | `'EXE'`

Output to build from generated C/C++ code, specified as one of the values in this table.

Value	Description
`'LIB'`	Static library

Value	Description
`'DLL'`	Dynamically linked library
`'EXE'`	Executable program

`PassStructByReference` — Pass structures by reference
`true` (default) | `false`

Pass structures by reference to entry-point functions, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The generated code passes structures by reference, which reduces memory usage and execution time by minimizing the number of copies of parameters at entry-point function boundaries. **Note** An entry-point function that writes to a field of a structure parameter overwrites the input value.
`false`	The generated code passes structures by value.

This parameter applies only to entry-point functions.

See “Pass Structure Arguments by Reference or by Value in Generated Code”.

`PostCodeGenCommand` — Command to customize build processing
`''` (default) | character vector

Command to customize build processing after MEX function generation with `codegen`, specified as a character vector.

See “Build Process Customization”.

`PreserveArrayDimensions` — N-dimensional indexing
`false` (default) | `true`

Generation of code that uses N-dimensional indexing, specified as one of the values in this table.

Value	Description
`false`	Generate code that uses one-dimensional indexing. (This value is the default value.)
`true`	Generate code that uses N-dimensional indexing.

See “Generate Code That Uses N-Dimensional Indexing”.

`PreserveVariableNames` — Variable names to preserve in the generated code
`'None'` (default) | `'UserNames'` | `'All'`

Variable names to preserve in the generated code, specified as one of the values in this table.

Value	Description
`'None'`	This value is the default value. The code generator does not have to preserve any variable names. It can reuse any variables that meet the requirements for variable reuse. If your code uses large structures or arrays, setting `PreserveVariableNames` to `'None'` can reduce memory usage or improve execution speed.

Value	Description
`'UserNames'`	The code generator preserves names that correspond to variables that you define in the MATLAB code. It does not replace your variable name with another name and does not use your name for another variable. To improve readability, set `PreserveVariableNames` to `'UserNames'`. Then, you can more easily trace the variables in the generated code back to the variables in your MATLAB code. Setting `PreserveVariableNames` to `'UserNames'` does not prevent an optimization from removing your variables from the generated code or prevent the C/C++ compiler from reusing the variables in the generated binary code.
`'All'`	Preserve all variable names. This parameter value disables variable reuse. Use it only for testing or debugging, not for production code.

See “Preserve Variable Names in Generated Code”.

`ReportInfoVarName` — Name of variable containing code generation report information

`''` (default) | character vector

Name of variable to which you export information about code generation, specified as a character vector. The code generator creates this variable in the base MATLAB workspace. This variable contains information about code generation settings, input files, generated files, and code generation messages.

See “Access Code Generation Report Information Programmatically” and coder.ReportInfo Properties.

`ReportPotentialDifferences` — Potential differences reporting

`true` (default) | `false`

Potential difference reporting, specified as one of the values in this table:

Value	Description
`true`	The code generator reports potential behavior differences between generated code and MATLAB code. The potential differences are listed on a tab of the code generation report. A potential difference is a difference that occurs at run time only under certain conditions.
`false`	The code generator does not report potential differences.

See “Potential Differences Reporting”.

`ReservedNameArray` — Names that the code generator cannot use for functions or variables
`''` (default) | character vector

Names that code generator cannot use for functions or variables, specified as a character vector.

`RowMajor` — Row-major array layout
`false` (default) | `true`

Generation of code that uses row-major array layout, specified as one of the values in this table.

Value	Description
`false`	Generate code that uses column-major array layout. (This value is the default value.)
`true`	Generate code that uses row-major array layout.

See “Generate Code That Uses Row-Major Array Layout”.

`RuntimeChecks` — Run-time error detection and reporting in generated code
`false` (default) | `true`

Run-time error detection and reporting in generated code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The generated code does not check for errors such as out-of-bounds array indexing.
`true`	The generated code checks for errors such as out-of-bounds array indexing. The error-reporting software uses `fprintf` to write error messages to `stderr`. It uses `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. The `abort` function abruptly terminates the program. If your system supports signals, you can catch the abort signal (`SIGABRT`) so that you can control the program termination. Error messages are in English.

See “Run-Time Error Detection and Reporting in Standalone C/C++ Code”.

`SaturateOnIntegerOverflow` — Integer overflow support
`true` (default) | `false`

Integer overflow support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator produces code to handle integer overflow. Overflows saturate to either the minimum or maximum value that the data type can represent.

Value	Description
`false`	The code generator does not produce code to handle integer overflow. Do not set `SaturateOnIntegerOverflow` to `false` unless you are sure that your code does not depend on integer overflow support. If you disable integer overflow support and run-time error checking is enabled, the generated code produces an error for overflows. If you disable integer overflow support and you disable run-time error checking, the overflow behavior depends on your target C compiler. In the C standard, the behavior for integer overflow is undefined. However, most C compilers wrap on overflow.

This parameter applies only to MATLAB built-in integer types. It does not apply to doubles, singles, or fixed-point data types.

See “Disable Support for Integer Overflow or Nonfinites”.

`StackUsageMax` — Maximum stack usage per application
`200000` (default) | positive integer

Maximum stack usage per application, in bytes, specified as a positive integer. Set a limit that is lower than the available stack size. Otherwise, a run-time stack overflow might occur. The C compiler detects and reports stack overflows.

See “Disable Support for Integer Overflow or Nonfinites”.

`SupportNonFinite` — Support for nonfinite values
`true` (default) | `false`

Support for nonfinite values, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If `GenerateNonFiniteFilesIfUsed` is set to `true`, the code generator produces code to support nonfinite values (`Inf` and `NaN`) only if they are used. If `GenerateNonFiniteFilesIfUsed` is set to `false`, the code generator always produces code to support nonfinite values (`Inf` and `NaN`).
`false`	The code generator does not produce code to support nonfinite values.

See “Disable Support for Integer Overflow or Nonfinites”.

`TargetLang` — Language to use in generated code
`'C'` (default) | `'C++'`

Language to use in generated code, specified as `'C'` or `'C++'`. If you specify C++, the code generator wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Dependency: If `DeepLearningConfig` is set, `codegen` sets `TargetLang` to `C++`.

`TargetLangStandard` — Standard math library to use for the generated code
`'C89/C90 (ANSI)'` | `'C99 (ISO)'` | `'C++03 (ISO)'`

Standard math library to use for the generated code, specified as one of these character vectors:

- `'C89/C90 (ANSI)'`
- `'C99 (ISO)'`
- `'C++03 (ISO)'`

The code generator uses the standard math library for calls to math operations. The default standard math library depends on the language that you select. For C, the default library is `'C99 (ISO)'`. For C++, the default library is `'C++03 (ISO)'`.

See “Change the Standard Math Library”.

`Toolchain` — Toolchain to use for building a C/C++ library or executable program
`'Automatically locate an installed toolchain'` (default) | character vector

Toolchain to use for building a C/C++ library or executable program, specified as a character vector. The list of available toolchains depends on the host computer platform, and can include custom toolchains that you added. If you do not specify a toolchain, the code generator locates an installed toolchain.

Note:

- Suppose that you create a `coder.CodeConfig` object `cfg` in a MATLAB session and use it in another MATLAB session. If the MATLAB host computer for the second session does not have the toolchain specified in the `cfg.Toolchain` property installed on it, this parameter reverts to its default value. The default value is `'Automatically locate an installed toolchain'`.

`Verbose` — Code generation progress display
false (default) | `true`

Code generation progress display, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not display code generation progress.
`true`	The code generator displays code generation progress, including code generation stages and compiler output.

Examples

Specify Configuration Parameters for Generation of Standalone Code

Write a MATLAB function from which you can generate code. This example uses the function `myadd` that returns the sum of its inputs.

```
function c = myadd(a,b)
c = a + b;
end
```

Create a configuration object for generation of standalone C/C++ code (a static library, a dynamically linked library, or an executable program). For example, create a configuration object for generation of a static library.

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

Change the values of the properties for which you do not want to use the default values. For example, enable run-time error detection and reporting in the generated C/C++ code.

```
cfg.RuntimeChecks = true;
```

Generate code by using `codegen`. Pass the configuration object to `codegen` by using the `-config` option. Specify that the input arguments are scalar double.

```
codegen myadd -config cfg -args {1 1} -report
```

Alternative Functionality

To use default configuration parameters for build types `'LIB'`, `'DLL'`, or `'EXE'`, use the `codegen` option `-config:lib`, `-config:dll`, or `-config:exe`, respectively. Then, you do not have to create and pass a configuration object to `codegen`.

See Also

Functions
`codegen` | `coder.config` | `pathsep`

Objects
`coder.EmbeddedCodeConfig` | `coder.HardwareImplementation` | `coder.MexCodeConfig`

Topics

“Generating Standalone C/C++ Executables from MATLAB Code”
“Generating C/C++ Static Libraries from MATLAB Code”
“Generating C/C++ Dynamically Linked Libraries from MATLAB Code”

Introduced in R2011a

coder.EmbeddedCodeConfig

Configuration parameters for C/C++ code generation from MATLAB code with Embedded Coder

Description

A `coder.EmbeddedCodeConfig` object contains the configuration parameters that `codegen` uses for generating a static library, a dynamically linked library, or an executable program with Embedded Coder. Pass the object to the `codegen` function by using the `-config` option.

Creation

Syntax

```
cfg = coder.config(build_type)
cfg = coder.config(build_type, 'ecoder', true)
```

Description

`cfg = coder.config(build_type)` creates a code generation configuration object for the specified build type, which can be a static library, a dynamically linked library, or an executable program. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` object.

`cfg = coder.config(build_type, 'ecoder', true)` creates a `coder.EmbeddedCodeConfig` object for the specified output type even if the Embedded Coder product is not installed. However, you cannot generate code using a `coder.EmbeddedCodeConfig` object unless you have Embedded Coder installed.

Input Arguments

`build_type` — Output to build from generated C/C++ code

`'LIB'` | `'DLL'` | `'EXE'`

Output to build from generated C/C++ code, specified as one of the values in this table.

Value	Description
`'LIB'`	Static library
`'DLL'`	Dynamically linked library
`'EXE'`	Executable program

Properties

`BuildConfiguration` — Compiler optimization or debug settings for toolchain

`'Faster Builds'` (default) | `'Faster Runs'` | `'Debug'` | `'Specify'`

Compiler optimization or debug settings for toolchain, specified as one of the values in this table.

Value	Description
`'Faster Builds'`	Optimizes the build for shorter build times.
`'Faster Runs'`	Optimizes the build for faster running executables.
`'Debug'`	Optimizes the build for debugging.
`'Specify'`	Enables the `CustomToolchainOptions` property for customization of settings for tools in the selected toolchain. If the `Toolchain` property is set to `'Automatically locate an installed toolchain'`, then setting `BuildConfiguration` to `'Specify'` changes `Toolchain` to the located toolchain.

`CastingMode` — Data type casting level

`'Nominal'` (default) | `'Standards'` | `'Explicit'`

Data type casting level for variables in the generated C/C++ code, specified as one of the values in this table.

Value	Description

`'Nominal'`	This value is the default value.
	
	Generates C/C++ code that uses default C compiler data type casting. For example:
	
	`short addone(short x)`
	`{`
	`  int i0;`
	`  i0 = x + 1;`
	`  if (i0 > 32767) {`
	`  i0 = 32767;`
	`  }`
	
	`  return (short)i0;`
	`}`

`'Standards'`	Generates C/C++ code that casts data types to conform to MISRA standards. For example:
	
	`short addone(short x)`
	`{`
	`  int i0;`
	`  i0 = (int)x + (int)1;`
	`  if (i0 > (int)32767) {`
	`  i0 = (int)32767;`
	`  }`
	
	`  return (short)i0;`
	`}`

Value	Description

`'Explicit'`	Generates C/C++ code that casts data type values explicitly. For example:
	`short addone(short x)`
	`{`
	`  int i0;`
	`  i0 = (int)x + 1;`
	`  if (i0 > 32767) {`
	`  i0 = 32767;`
	`  }`
	`  return (short)i0;`
	`}`

See “Control Data Type Casts in Generated Code” (Embedded Coder).

`CodeExecutionProfiling` — Execution time profiling during a SIL or PIL execution
`false` (default) | `true`

Execution-time profiling during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. Disables execution-time profiling during a SIL or PIL execution.
`true`	Enables execution-time profiling during a SIL or PIL execution.

See “Execution Time Profiling for SIL and PIL” (Embedded Coder).

`CodeTemplate` — Code generation template for file and function banners
`[]` (default) | `coder.MATLABCodeTemplate` object

Code generation template for file and function banners in the generated code. By default, `CodeTemplate` is empty and the code generator produces default banners. To produce custom banners, set `CodeTemplate` to a `coder.MATLABCodeTemplate` object created

from a code generation template (CGT) file. See “Generate Custom File and Function Banners for C/C++ Code” (Embedded Coder).

`CodeReplacementLibrary` — Code replacement library for generated code
character vector

Code replacement library for generated code, specified as one of the values in this table:

Value	Description
`'None'`	This value is the default value. Does not use a code replacement library.
Named code replacement library	Generates calls to a specific platform, compiler, or standards code replacement library. The list of named libraries depends on: • Installed support packages. • System target file, language, standard math library, and device vendor configuration. • Whether you created and registered code replacement libraries, using the Embedded Coder product.

Compatible libraries depend on these parameters:

- `TargetLang`
- `TargetLangStandard`
- `ProdHWDeviceType` in the hardware implementation configuration object.

Embedded Coder offers more libraries and the ability to create and use custom code replacement libraries.

MATLAB Coder generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Note MATLAB Coder software does not support TLC callbacks.

`ColumnLimit` — Maximum number of columns before a line break in the generated code

80 (default) | positive integer

Maximum number of columns before a line break in the generated code, specified as a positive integer in the range [45, 65536].

Other rules for placement of the line break can take precedence over the column limit that you specify.

`CommentStyle` — Comment style in the generated code

`'Auto'` (default) | `'Single-line'` | `'Multi-line'`

Comment style in the generated code, specified as one of the values in this table.

Value	Description
`'Auto'`	For C, generate multiline comments. For C++, generate single-line comments.
`'Single-line'`	Generate single-line comments preceded by `//`.
`'Multi-line'`	Generate single or multiline comments delimited by `/*` and `*/`.

For C code generation, specify the single-line comment style only if your compiler supports it.

Dependency: `GenerateComments` enables this parameter.

See “Specify Comment Style for C/C++ Code” (Embedded Coder).

`CompileTimeRecursionLimit` — Maximum number of function specializations for compile-time recursion

50 (default) | positive integer

Maximum number of function specializations for compile-time recursion, specified as a positive integer. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to 0. The default compile-time recursion limit is large enough for most recursive functions that require this type of recursion. If code generation fails because of the compile-time recursion limit, and you want compile-time recursion,

try to increase the limit. Alternatively, change your MATLAB code so that the code generator uses run-time recursion. See “Compile-Time Recursion Limit Reached”.

`ConstantFoldingTimeout` — Maximum number of instructions to be executed by the constant folder

40000 (default) | positive integer

Maximum number of instructions that the constant folder executes. In some situations, code generation requires specific instructions to be constant. If constant folding stops before these instructions are constant-folded, code generation fails. In this case, increase the value of `ConstantFoldingTimeout`.

See “MATLAB Coder Optimizations in Generated Code”.

`ConvertIfToSwitch` — Conversion of `if-elseif-else` patterns to `switch-case` statements

`false` (default) | `true`

Conversion of `if-elseif-else` patterns to `switch-case` statements in the generated code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not convert `if-elseif-else` patterns to `switch-case` statements.
`true`	The code generator tries to convert `if-elseif-else` patterns to `switch-case` statements. The code generator produces a `switch-case` statement only if all potential case expressions are scalar integer values.

See “Controlling C Code Style” (Embedded Coder).

`CustomBLASCallback` — BLAS callback class

' ' (default) | character vector

Callback class for BLAS library calls in code generated for certain low-level vector and matrix operations in MATLAB code, specified as a character vector.

If you specify a BLAS callback class, for certain low-level vector and matrix functions, the code generator produces BLAS calls by using the CBLAS C interface to your BLAS library. The callback class provides the name of your CBLAS header file, the names of CBLAS data types, and the information required to link to your BLAS library. If this parameter is empty, the code generator produces code for matrix functions instead of a BLAS call.

See “Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”.

`CustomFFTCallback` — Callback class for FFTW library calls
' ' (default) | character vector

Callback class for FFTW library calls in code generated for FFT functions in MATLAB code, specified as a character vector.

To improve the execution speed of FFT functions, the code generator produces calls to the FFTW library that you specify in the callback class. If this parameter is empty, the code generator uses its own algorithms for FFT functions instead of calling the FFTW library.

See “Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”.

`CustomHeaderCode` — Custom code that appears at top of generated C/C++ header files
' ' (default) | character vector

Custom code that appears near the top of each C/C++ header file generated from your MATLAB code, specified as a character vector.

`CustomInclude` — Include folders to add to include path for compiling generated code
' ' (default) | character vector

Include folders to add to the include path when compiling the generated code. Specify the list of include folders as a character vector. In the character vector, separate include folders by a `pathsep` character. For example:

```
cfg = coder.config('lib','ecoder',true);
cfg.CustomInclude = ['C:\Project' pathsep 'C:\Custom Files'];
```

`CustomInitializer` — Custom code to include in the generated initialize function
' ' (default) | character vector

Custom code to include in the generated initialize function, specified as a character vector.

`CustomLAPACKCallback` — LAPACK callback class

' ' (default) | character vector

Callback class for LAPACK library calls in code generated for certain linear algebra functions in MATLAB code, specified as a character vector.

If you specify a LAPACK callback class, for certain linear algebra functions, the code generator produces LAPACK calls by using the LAPACKE C interface to your LAPACK library. The callback class provides the name of your LAPACKE header file and the information required to link to your LAPACK library. If this parameter is empty, the code generator produces code for linear algebra functions instead of a LAPACK call.

See “Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”.

`CustomLibrary` — Static library files to link with the generated code

' ' (default) | character vector

Static library files to link with the generated code, specified as a character vector. In the character vector, separate library file names by a `pathsep` character.

`CustomSource` — Source files to compile and link with the generated code

' ' (default) | character vector

Source files to compile and link with the generated code, specified as a character vector. In the character vector, separate source file names by a `pathsep` character.

The build process searches for the source files first in the current folder, and then in the include folders that you specify in `CustomInclude`. If source files with the same name occur in multiple folders on the search path, the build process might use a different file than the file that you specified.

Suppose that you specify `foo.cpp` as a source file. If `foo.c` and `foo.cpp` are both on the search path, you cannot be sure whether the build process uses `foo.c` or `foo.cpp`.

`CustomSourceCode` — Code to appear near the top of the generated `.c` or `.cpp` file

' ' (default) | character vector

Specify code to appear near the top of the generated `.c` or `.cpp` file, outside of any function. Specify code as a character vector.

Do not specify a C static function definition.

`CustomSymbolStrEMXArray` — Custom identifier format for EMX array types
`'emxArray_$M$N'` (default) | character vector

Custom identifier format for generated EMX Array types (Embeddable mxArray types), specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrEMXArrayFcn` — Custom identifier format for EMX array utility functions
`'emx$M$N'` (default) | character vector

Custom identifier format for generated EMX Array (Embeddable mxArrays) utility functions, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrFcn` — Custom identifier format for local function identifiers
`'m_$M$N'` (default) | character vector

Custom identifier format for generated local function identifiers, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrField` — Custom identifier format for field names in global type identifiers
`'$M$N'` (default) | character vector

Custom identifier format for generated field names in global type identifiers, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrGlobalVar` — Custom identifier format for global variable identifiers
`'$M$N'` (default) | character vector

Custom identifier format for generated global variable identifiers, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrMacro` — Custom identifier format for constant macro identifiers
`'$M$N'` (default) | character vector

Custom identifier format for generated constant macro identifiers, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrTmpVar` — Custom identifier format for local temporary variable identifiers
`'$M$N'` (default) | character vector

Custom identifier format for generated local temporary variable identifiers, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomSymbolStrType` — Custom identifier format for global type identifiers
`'$M$N'` (default) | character vector

Custom identifier format for generated global type identifiers, specified as a character vector. To specify the format, see “Customize Generated Identifiers” (Embedded Coder).

`CustomTerminator` — Code that appears in the generated terminate function
`''` (default) | character vector

Code that appears in the generated terminate function, specified as a character vector.

`CustomToolchainOptions` — Custom settings for tools in selected toolchain
cell array

Custom settings for tools in selected toolchain, specified as a cell array.

Dependencies:

- The `Toolchain` property determines which tools and options appear in the cell array.
- Setting the `BuildConfiguration` property to `Specify` enables `CustomToolchainOptions`.

First, get the current settings. For example:

```
cfg = coder.config('lib');
cfg.BuildConfiguration='Specify';
opt = cfg.CustomToolchainOptions
```

Then, edit the values in `opt`.

These values derive from the toolchain definition file and the third-party compiler options. See “Custom Toolchain Registration”.

`DataTypeReplacement` — Data type replacement in generated code
`'CBuiltIn'` | `'CoderTypeDefs'`

Data type replacement in generated code, specified as one of the values in this table.

Value	Description
`'CBuiltIn'`	This value is the default value. The code generator uses built-in C data types.
`'CoderTypeDefs'`	The code generator uses predefined data types from `rtwtypes.h`

`DeepLearningConfig` — Configuration object for deep learning code generation
`coder.MklDNNConfig` object | `coder.ARMNEONConfig` object

Configuration object for code generation for deep learning networks, specified as a `coder.MklDNNConfig` object or a `coder.ARMNEONConfig` object.

A `coder.MklDNNConfig` object contains parameters specific to C++ code generation for deep learning using Intel MKL-DNN. To create a `coder.MklDNNConfig` object, use `coder.DeepLearningConfig`. For example:

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

A `coder.ARMNEONConfig` object contains parameters specific to C++ code generation for deep learning using the ARM Compute Library. To create a `coder.ARMNEONConfig` object, use `coder.DeepLearningConfig`. For example:

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('arm-compute');
```

Code generation for deep learning networks requires Deep Learning Toolbox and the MATLAB Coder Interface for Deep Learning Libraries support package.

See “Code Generation for Deep Learning Networks with MKL-DNN” and “Code Generation for Deep Learning Networks with ARM Compute Library”.

Dependency: If `DeepLearningConfig` is set, `codegen` sets `TargetLang` to `C++`.

`Description` — Object description
`'class CodeConfig: C code generation configuration.'` (default) | character vector

Object description, specified as a character vector.

`DynamicMemoryAllocation` — Dynamic memory allocation mode
`'Threshold'` (default) | `'AllVariableSizeArrays'` | `'Off'`

Dynamic memory allocation mode, specified as one of the values in this table.

Value	Description
`'Threshold'`	This value is the default value. The code generator allocates memory dynamically on the heap for variable-size arrays whose size (in bytes) is greater than or equal to `DynamicMemoryAllocationThreshold`.
`'AllVariableSizeArrays'`	The code generator dynamically allocates memory for all variable-size arrays on the heap.
`'Off'`	The code generator statically allocates memory for variable-size arrays on the stack.

Unbounded variable-size arrays require dynamic memory allocation.

Dependencies:

- `EnableVariableSizing` enables this parameter.
- Setting this `DynamicMemoryAllocation` to `'Threshold'` enables the `DynamicMemoryAllocationThreshold` parameter.

See “Generate Code for Variable-Size Data”.

`DynamicMemoryAllocationThreshold` — Size threshold for dynamic memory allocation of variable-size arrays
`65536` (default) | positive integer

Size threshold for dynamic memory allocation of variable-size arrays, specified as a positive integer. The code generator uses dynamic memory allocation for variable-size arrays whose size (in bytes) is greater than or equal to the threshold.

Dependency:

- Setting `DynamicMemoryAllocation` to `'Threshold'` enables this parameter.

See “Generate Code for Variable-Size Data”.

`EnableAutoExtrinsicCalls` — Automatic extrinsic function calls
`true` (default) | `false`

Automatic extrinsic function calls, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator treats some common visualization functions as extrinsic functions. You do not have to declare these functions as extrinsic by using `coder.extrinsic`. This capability reduces the amount of time that you spend making your code suitable for code generation.
`false`	The code generator does not treat common visualization functions as extrinsic functions unless you declare them as extrinsic by using `coder.extrinsic`

Some common visualization functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions”.

`EnableCustomReplacementTypes` — Custom names for data types in generated code
`false` (default) | `true`

Custom names for MATLAB data types in generated C/C++ code, specified as one of the values in the table.

Value	Description
`false`	This value is the default value. Custom names for the MATLAB data types are not allowed.
`true`	Custom names for the MATLAB data types are allowed. Specify custom names by using `ReplacementTypes`. Setting `EnableCustomReplacementTypes` to `true` enables the `ReplacementTypes` parameter.

`EnableMemcpy` — `memcpy` optimization
`true` (default) | `false`

`memcpy` optimization, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the `memcpy` optimization. To optimize code that copies consecutive array elements, the `memcpy` optimization replaces the code with a `memcpy` call. When the number of elements to copy is known at compile time, the code generator uses the `MemcpyThreshold` property to determine whether to use the optimization. See “memcpy Optimization”.
`false`	The code generator does not use the `memcpy` optimization.

`EnableOpenMP` — Parallelization of `parfor`-loops
`true` (default) | `false`

Parallelization of `parfor`-loops, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the OpenMP library to produce loop iterations that run in parallel.
`false`	The code generator treats `parfor`-loops as `for`-loops.

See `parfor`.

Use of the OpenMP library is not compatible with just-in-time (JIT) compilation. If `EnableJIT` and `EnableOpenMP` are `true`, the code generator uses JIT compilation and treats `parfor`-loops as `for`-loops.

`EnableRuntimeRecursion` — Run-time recursion support
`true` (default) | `false`

Run-time recursion support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Recursive functions are allowed in the generated code.
`false`	Recursive functions are not allowed in the generated code.

Some coding standards, such as MISRA, do not allow recursion. To increase the likelihood of generating code that is compliant with MISRA C, set `EnableRuntimeRecursion` to `false`.

If your MATLAB code requires run-time recursion and `EnableRuntimeRecursion` is `false`, code generation fails.

See “Code Generation for Recursive Functions”.

`EnableSignedLeftShifts` — Replacement of multiplications by powers of two with signed left bitwise shifts
`true` (default) | `false`

Replacement of multiplications by powers of two with signed left bitwise shifts in the generated C/C++ code, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator uses signed left shifts for multiplication by powers of two. An example of generated C code that uses signed left shift for multiplication by eight is: `i <<= 3;`
`false`	The code generator does not use signed left shifts for multiplication by powers of two. An example of generated C code that does not use signed left shift for multiplication by eight is: `i = i * 8;`

Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. To increase the likelihood of generating MISRA C compliant code, set `EnableSignedLeftShifts` to `false`.

See “Control Signed Left Shifts in Generated Code” (Embedded Coder).

`EnableSignedRightShifts` — Signed right bitwise shifts in generated code
`true` (default) | `false`

Signed right bitwise shifts in generated code, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator uses signed right shifts. An example of generated C code that uses a signed right shift is: `i >>= 3`

Value	Description
`false`	The code generator replaces right shifts on signed integers with a function call in the generated code. For example: `i = asr_s32(i, 3U);`

Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. To increase the likelihood of generating MISRA-C:2004 compliant code, set `EnableSignedRightShifts` to `false`.

`EnableStrengthReduction` — Strength reduction optimization
`false` (default) | `true`

Strength reduction optimization, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not use the strength reduction optimization.
`true`	The code generator tries to use the strength reduction optimization to simplify array indexing in loops in the generated code. When possible, for array indices in loops, the code generator replaces multiply operations with add operations. Multiply operations can be expensive. When the C/C++ compiler on the target platform does not optimize the array indexing, the strength reduction optimization is useful. Even when the optimization replaces the multiply operations in the generated code, it is possible that the C/C++ compiler can generate multiply instructions.

`EnableTraceability` — Traceability in code generation report
true (default) | false

Traceability in code generation report, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generation report includes code traceability. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).
`false`	The code generation report does not include code traceability.

`EnableVariableSizing` — Variable-size array support
`true` (default) | `false`

Variable-size array support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Variable-size arrays are allowed for code generation.
`false`	Variable-size arrays are not allowed for code generation.

Dependency:

- Enables `Dynamic memory allocation`.

See “Code Generation for Variable-Size Arrays”.

`FilePartitionMethod` — File partitioning mode
`'MapMFileToCFile'` (default) | `'SingleFile'`

File partitioning mode specified as one of the values in this table.

Value	Description
`'MapMFileToCFile'`	This value is the default value. The code generator produces separate C/C++ files for each MATLAB language file.

Value	Description
`'SingleFile'`	The code generator produces a single file for C/C++ functions that map to your MATLAB entry-point functions. The code generator produces separate C/C++ files for utility functions.

See “How MATLAB Coder Partitions Generated Code”.

`GenerateCodeMetricsReport` — Static code metrics report
`false` (default) | `true`

Static code metrics report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not run static code metrics analysis at code generation time. You can run the analysis and produce the report later by clicking **Code Metrics** on the **Summary** tab of the code generation report.
`true`	The code generator runs static code metrics analysis and produces the report at code generation time.

To open a code metrics report, click the **Code Metrics** link on the **Summary** tab of the code generation report.

Dependency:

- The code generator produces a static code metrics report only if **GenerateReport** is `true` or if you specify the `-report` option of the `codegen` report.

See “Generating a Static Code Metrics Report for Code Generated from MATLAB Code” (Embedded Coder).

`GenCodeOnly` — Generation of only source code
`false` (default) | `true`

Generation of only source code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator produces C/C++ source code and builds object code.
`true`	The code generator produces C/C++ source code, but does not invoke the make command or build object code. When you iterate between modifying MATLAB code and generating C/C++ code, generating only code can save time.

`GenerateCodeReplacementReport` — Code replacement report
`false` (default) | `true`

Code replacement report, specified as on of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not produce a code replacements report.
`true`	The code generator produces a code replacements report that summarizes the replacements from the selected code replacement library. The report provides a mapping between each code replacement instance and the line of MATLAB code that triggered the replacement.

`GenerateComments` — Comments in generated code
`true` (default) | `false`

Comments in generated code, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator places comments in the generated code.
`false`	The code generator does not place comments in the generated code.

`GenerateDefaultInSwitch` — Default case for all switch statements
`false` (default) | `true`

Default case for all switch statements, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator might not generate a default case for some switch statements.
`true`	The code generator produces a default case for all switch statements in the generated code.

Some coding standards, such as MISRA, require the default case for switch statements.

`GenerateExampleMain` — Example C/C++ main file generation
`'GenerateCodeOnly'` (default) | `'DoNotGenerate'` | `'GenerateCodeAndCompile'`

Example C/C++ main file generation, specified as one of the values in this table.

Value	Description
`'GenerateCodeOnly'`	This value is the default value. The code generator generates an example C/C++ main function but does not compile it.
`'DoNotGenerate'`	The code generator does not generate an example C/C++ main function.

Value	Description
`'GenerateCodeAndCompile'`	The code generator generates an example C/C++ main function and compiles it to create a test executable. This executable does not return output. If the `GenCodeOnly` parameter is `true`, the code generator does not compile the C/C++ main function.

An example main function is a template to help you to write a C/C++ main function that calls generated C/C++ code. See “Incorporate Generated Code Using an Example Main Function”.

`GenerateMakefile` — Makefile generation
`true` (default) | `false`

Makefile generation during the build process, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator generates a makefile during the build process.
`false`	The code generator does not generate a makefile during the build process. Specify instructions for post-code-generation processing, including compilation and linking, in a post-code-generation command. See “Build Process Customization”.

`GenerateNonFiniteFilesIfUsed` — Generate support files for nonfinite data only if nonfinite data is used
`true` (default) | `false`

Generation of support files for nonfinite data, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator produces the support files for nonfinite data (`Inf` and `NaN`) only if the generated code uses nonfinite data.
`false`	The code generator always produces the support files for nonfinite data (`Inf` and `NaN`).

Dependency:

- Setting `SupportNonFinite` to `true` enables this parameter.

`GenerateReport` — Code generation report
`false` (default) | true

Code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator produces a report only if error or warning messages occur, or if you set `LaunchReport` to `true`.
`true`	The code generator produces a code generation report.

`Hardware` — Object that specifies a hardware board
`coder.Hardware` object

Object that specifies a hardware board. To create the `coder.Hardware` object, use `coder.hardware`. For example:

```
cfg = coder.config('lib');
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Before you use `coder.hardware`, you must install the support package for the hardware board.

Dependencies:

- Setting `Hardware` customizes the hardware implementation object and other configuration parameters for a particular hardware board.
- If `DeepLearningConfig` is set to a `coder.ARMNEONConfig` object and `Hardware` is empty, then `codegen` sets the `GenCodeOnly` property to `true`.

Note:

- Suppose that you create a `coder.CodeConfig` object `cfg` in a MATLAB session and use it in another MATLAB session. If the MATLAB host computer for the second session does not have the hardware board specified in the `cfg.Hardware` property installed on it, this parameter reverts to its default value. The default value is `[]`.

To specify a hardware board for PIL execution, see “PIL Execution with ARM Cortex-A at the Command Line” (Embedded Coder).

`HardwareImplementation` — Hardware implementation object
`coder.HardwareImplementation` object

Hardware implementation object that specifies hardware-specific configuration parameters for C/C++ code generation. `coder.config` creates a `coder.EmbeddedCodeConfig` object with the `HardwareImplementation` property set to a `coder.HardwareImplementation` object with default parameter values for the MATLAB host computer.

`HighlightPotentialDataTypeIssues` — Highlighting of potential data type issues in the code generation report
`false` (default) | `true`

Highlighting of potential data type issues in the code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. This code generation report does not highlight potential data type issues.

Value	Description
`true`	The code generation report highlights MATLAB code that results in single-precision or double-precision operations in the generated C/C++ code. If you have Fixed-Point Designer, the report also highlights expressions in the MATLAB code that result in expensive fixed-point operations in the generated code.

`HighlightPotentialRowMajorIssues` — Potential row-major layout issues
`true` (default) | `false`

Display of potential row-major layout efficiency issues, specified as one of the values in this table.

Value	Description
`true`	The code generation report displays potential efficiency issues due to row-major layout. (This value is the default value.)
`false`	The code generation report does not display issues related to array layout.

See “Code Design for Row-Major Array Layout”.

`IncludeTerminateFcn` — Terminate function generation
`true` (default) | `false`

Terminate function generation, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator produces a terminate function.

Value	Description
`false`	The code generator does not produce a terminate function. If you set `IncludeTerminateFcn` to `false` and a terminate function is required, for example, to free memory, the code generator issues a warning.

`IndentSize` — Number of characters per indentation level

2 (default) | positive integer

Number of characters per indentation level, specified as a positive integer in the range [2,8].

`IndentStyle` — Style for placement of braces in the generated code

`'K&R'` (default) | `'Allman'`

Style for placement of braces in the generated code, specified as one of the values in this table.

Value	Description

`'K&R'`	This value is the default value.
	
	For blocks within a function, an opening brace is on the same line as its control statement. For example:
	
	`void addone(const double x[6], double z[6])`
	`{`
	`  int i0;`
	`  for (i0 = 0; i0 < 6; i0++) {`
	`  z[i0] = x[i0] + 1.0;`
	`  }`
	`}`

Value	Description

`'Allman'`	For blocks within a function, an opening brace is on its own line at the same indentation level as its control statement. For example:
	`void addone(const double x[6], double z[6])`
	`{`
	`  int i0;`
	`  for (i0 = 0; i0 < 6; i0++)`
	`  {`
	`  z[i0] = x[i0] + 1.0;`
	`  }`
	`}`

`InitFltsAndDblsToZero` — Assignment of float and double zero with memset
`true` (default) | `false`

Assignment of float and double zero with `memset`, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the `memset` optimization for assignment of floating-point zero to consecutive array elements. To assign consecutive array elements, the `memset` optimization uses a `memset` call. When the number of elements to assign is known at compile time, the code generator uses the `MemcpyThreshold` property to determine whether to use the optimization. See “memset Optimization”.
`false`	The code generator does not use the `memset` optimization for assignment of float and double zero to consecutive array elements.

`InlineStackLimit` — Stack size limit for inlined functions
4000 (default) | positive integer

Stack size limit for inlined functions, specified as a positive integer. The stack size limit determines the amount of stack space allocated for local variables of the inlined function.

Specifying a limit for the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even when the function returns.

This capability is especially important for embedded processors where stack size can be limited.

See “Control Inlining”.

`InlineThreshold` — Function size threshold for inlining
10 (default) | positive integer

Function size threshold for inlining, specified as a positive integer.

Unless other conditions prevent inlining, the code generator inlines functions that are smaller than the threshold.

The function size is measured as an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. To obtain the inlining behavior that you want, experiment with the threshold. For example, if the default threshold results in inlining of large functions and generation of large amounts of C code, tune the threshold until you are satisfied with the size of the generated code.

See “Control Inlining”.

`InlineThresholdMax` — Maximum size of functions after inlining
200 (default) | positive integer

Maximum size of functions after inlining, specified as a positive integer. If, after inlining, the size of the calling function exceeds `InlineThresholdMax`, then the code generator does not inline the called function.

The function size is measured as an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. To obtain the inlining behavior that you want, experiment with the threshold. For example, if the default threshold results in inlining of large functions and generation of large amounts of C code, tune the threshold until you are satisfied with the size of the generated code.

See “Control Inlining”.

`LaunchReport` — Automatic open of code generation report
`false` (default) | true

Automatic open of code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. If errors or warnings occur, or if `GenerateReport` is `true`, the code generator produces a report, but does not open the report.
`true`	The code generator produces and opens a code generation report.

`LoopUnrollThreshold` — Threshold on the number of iterations that determines whether to automatically unroll a `for`-loop
5 (default) | positive integer

Loops with fewer iterations than this threshold are candidates for automatic unrolling by the code generator. This threshold applies to all `for`-loops in your MATLAB code. For an individual `for`-loop, a `coder.unroll` directive placed immediately before the loop takes precedence over the loop unrolling optimization. The threshold can also apply to some `for`-loops produced during code generation.

See “Unroll for-Loops”.

`MATLABFcnDesc` — MATLAB function help text in function banner
`true` (default) | `false`

MATLAB function help text in function banner in generated code specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator includes MATLAB function help text in the function banner in the generated code.

Value	Description
`false`	The code generator treats the help text as a user comment.

If not selected, MATLAB Coder treats the help text as a user comment.

Dependencies:

- `GenerateComments` enables this parameter.

`MATLABSourceComments` — Inclusion of MATLAB source code as comments in generated code
`false` (default) | `true`

Inclusion of MATLAB source code as comments in generated code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not insert MATLAB source code as comments in the generated code. The code generator does not include the MATLAB function signature in the function banner.

Value	Description
`true`	The code generator inserts MATLAB source code as comments in the generated code. A traceability tag immediately precedes each line of source code. The traceability tag helps you to locate the corresponding MATLAB source code. See “Tracing Generated C/C++ Code to MATLAB Source Code”. If you have Embedded Coder, in the code generation report, the traceability tag links to the source code. The code generator also includes the MATLAB function signature in the function banner.

Dependency:

- `GenerateComments` enables this parameter.

See “Tracing Generated C/C++ Code to MATLAB Source Code”.

`MaxIdLength` — Maximum number of characters in generated identifiers
31 (default) | positive integer

Maximum number of characters in generated identifiers, specified as a positive integer in the range [31, 256]. This property applies to generated function, type definition, and variable names. To avoid truncation of identifiers by the target C compiler, specify a value that matches the maximum identifier length of the target C compiler.

This property does not apply to exported identifiers, such as the generated names for entry-point functions or emxArray API functions. If the length of an exported identifier exceeds the maximum identifier length of the target C compiler, the target C compiler truncates the exported identifier.

`MemcpyThreshold` — Minimum size for `memcpy` or `memset` optimization
64 (default) | positive integer

Minimum size, in bytes, for memcpy or memset optimization, specified as a positive integer.

To optimize generated code that copies consecutive array elements, the code generator tries to replace the code with a `memcpy` call. To optimize generated code that assigns a literal constant to consecutive array elements, the code generator tries to replace the code with a `memset` call.

The number of bytes is the number of array elements to copy or assign multiplied by the number of bytes required for the C/C++ data type.

If the number of elements to copy or assign is variable (not known at compile time), the code generator ignores the `MemcpyThreshold` property.

See “memcpy Optimization” and “memset Optimization”.

`MultiInstanceCode` — Multi-instance, reentrant code
`false` (default) | `true`

Multi-instance, reentrant code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not produce multi-instance, reentrant code.
`true`	The code generator produces reusable, multi-instance code that is reentrant.

See “Reentrant Code”.

`Name` — Object name
`'MexCodeConfig'` (default) | character vector

Object name, specified as a character vector.

`OutputType` — Output to build from generated C/C++ code
`'LIB'` | `'DLL'` | `'EXE'`

Output to build from generated C/C++ code, specified as one of the values in this table.

Value	Description
`'LIB'`	Static library

Value	Description
`'DLL'`	Dynamically linked library
`'EXE'`	Executable program

`ParenthesesLevel` — Parenthesization level in the generated code
`'Nominal'` (default) | `'Minimum'` | `'Maximum'`

Parenthesization level in the generated code, specified as one of the values in this table.

Value	Description
`'Nominal'`	This value is the default value. The code generator inserts parentheses to balance readability and visual complexity. For example: `Out = ((In2 - In1 > 1.0) && (In2 > 2.0));`
`'Maximum'`	The code generator includes parentheses to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA requirements. For example: `Out = (((In2 - In1) > 1.0) && (In2 > 2.0));`
`'Minimum'`	The code generator inserts parentheses where required by ANSI® C or C++, or to override default precedence. For example: `Out = In2 - In1 > 1.0 && In2 > 2.0;` If you generate C/C++ code that uses the minimum level, for certain settings in some compilers, you can receive compiler warnings. To eliminate these warnings, try the nominal level.

`PassStructByReference` — Pass structures by reference
`true` (default) | `false`

Pass structures by reference to entry-point functions, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The generated code passes structures by reference, which reduces memory usage and execution time by minimizing the number of copies of parameters at entry-point function boundaries. **Note** An entry-point function that writes to a field of a structure parameter overwrites the input value.
`false`	The generated code passes structures by value.

This parameter applies only to entry-point functions.

See “Pass Structure Arguments by Reference or by Value in Generated Code”.

`PostCodeGenCommand` — Command to customize build processing
`''` (default) | character vector

Command to customize build processing after MEX function generation with `codegen`, specified as a character vector.

See “Build Process Customization”.

`PreserveExternInFcnDecls` — Preservation of the `extern` keyword in function declarations
`true` (default) | `false`

Preservation of the `extern` keyword in function declarations in the generated code, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator includes the `extern` keyword in declarations for external functions.
`false`	The code generator removes the `extern` keyword from function declarations.

`PreserveArrayDimensions` — N-dimensional indexing
`false` (default) | `true`

Generation of code that uses N-dimensional indexing, specified as one of the values in this table.

Value	Description
`false`	Generate code that uses one-dimensional indexing. (This value is the default value.)
`true`	Generate code that uses N-dimensional indexing.

See “Generate Code That Uses N-Dimensional Indexing”.

`PreserveVariableNames` — Variable names to preserve in the generated code
`'None'` (default) | `'UserNames'` | `'All'`

Variable names to preserve in the generated code, specified as one of the values in this table.

Value	Description
`'None'`	This value is the default value. The code generator does not have to preserve any variable names. It can reuse any variables that meet the requirements for variable reuse. If your code uses large structures or arrays, setting `PreserveVariableNames` to `'None'` can reduce memory usage or improve execution speed.
`'UserNames'`	The code generator preserves names that correspond to variables that you define in the MATLAB code. It does not replace your variable name with another name and does not use your name for another variable. To improve readability, set `PreserveVariableNames` to `'UserNames'`. Then, you can more easily trace the variables in the generated code back to the variables in your MATLAB code. Setting `PreserveVariableNames` to `'UserNames'` does not prevent an optimization from removing your variables from the generated code or prevent the C/C++ compiler from reusing the variables in the generated binary code.
`'All'`	Preserve all variable names. This parameter value disables variable reuse. Use it only for testing or debugging, not for production code.

See “Preserve Variable Names in Generated Code”.

`PurelyIntegerCode` — Detection of floating-point code
`false` (default) | `true`

Detection of floating-point code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator allows floating-point data and operations.
`true`	The code generator does not allow floating-point data or operations. If the code generator detects floating-point data or operations, code generation ends with an error.

Dependency:

- Setting `PurelyIntegerCode` to `true` disables the `SupportNonFinite` parameter. Setting `PurelyIntegerCode` to `false` enables the `SupportNonFinite` parameter.

`ReplacementTypes` — Specify custom names for MATLAB data types
`''` (default) | `character vector`

Specify custom names for these MATLAB built-in data types: `double`, `single`, `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `char`, and `logical` that are in the generated C/C++ code.

Dependency:

- Setting `EnableCustomReplacementTypes` to `true` enables the `ReplacementTypes` parameter. See “Customize Data Type Replacement” (Embedded Coder).

`ReportInfoVarName` — Name of variable containing code generation report information
`''` (default) | character vector

Name of variable to which you export information about code generation, specified as a character vector. The code generator creates this variable in the base MATLAB workspace. This variable contains information about code generation settings, input files, generated files, and code generation messages.

See “Access Code Generation Report Information Programmatically” and coder.ReportInfo Properties.

`ReportPotentialDifferences` — Potential differences reporting
`true` (default) | `false`

Potential difference reporting, specified as one of the values in this table:

Value	Description
`true`	The code generator reports potential behavior differences between generated code and MATLAB code. The potential differences are listed on a tab of the code generation report. A potential difference is a difference that occurs at run time only under certain conditions.
`false`	The code generator does not report potential differences.

See “Potential Differences Reporting”.

`ReservedNameArray` — Names that the code generator cannot use for functions or variables

`''` (default) | character vector

Names that code generator cannot use for functions or variables, specified as a character vector.

`RowMajor` — Row-major array layout

`false` (default) | `true`

Generation of code that uses row-major array layout, specified as one of the values in this table.

Value	Description
`false`	Generate code that uses column-major array layout. (This value is the default value.)
`true`	Generate code that uses row-major array layout.

See “Generate Code That Uses Row-Major Array Layout”.

`RuntimeChecks` — Run-time error detection and reporting in generated code

`false` (default) | `true`

Run-time error detection and reporting in generated code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The generated code does not check for errors such as out-of-bounds array indexing.
`true`	The generated code checks for errors such as out-of-bounds array indexing. The error-reporting software uses `fprintf` to write error messages to `stderr`. It uses `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. The `abort` function abruptly terminates the program. If your system supports signals, you can catch the abort signal (`SIGABRT`) so that you can control the program termination. Error messages are in English.

See “Run-Time Error Detection and Reporting in Standalone C/C++ Code”.

`SaturateOnIntegerOverflow` — Integer overflow support
`true` (default) | `false`

Integer overflow support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator produces code to handle integer overflow. Overflows saturate to either the minimum or maximum value that the data type can represent.

Value	Description
`false`	The code generator does not produce code to handle integer overflow. Do not set `SaturateOnIntegerOverflow` to `false` unless you are sure that your code does not depend on integer overflow support. If you disable integer overflow support and run-time error checking is enabled, the generated code produces an error for overflows. If you disable integer overflow support and you disable run-time error checking, the overflow behavior depends on your target C compiler. In the C standard, the behavior for integer overflow is undefined. However, most C compilers wrap on overflow.

This parameter applies only to MATLAB built-in integer types. It does not apply to doubles, singles, or fixed-point data types.

See “Disable Support for Integer Overflow or Nonfinites”.

`SILDebugging` — Debugging of generated code during a SIL execution
`false` (default) | `true`

Source-level debugging of generated code during a software-in-the-loop (SIL) execution, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. Disables debugging of generated code during a SIL execution.

Value	Description
`true`	Enables the debugger to observe code behavior during a software-in-the-loop (SIL) execution. Supported debuggers: • On Windows, Microsoft Visual C++® debugger. • On Linux, GNU® Data Display Debugger (DDD).

See “Debug Generated Code During SIL Execution” (Embedded Coder).

`SILPILCheckConstantInputs` — Constant input checking mode for a SIL or PIL execution

`true` (default) | `false`

Constant input checking mode for a SIL or PIL execution, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The SIL or PIL execution compares the value that a test file provides for a constant input argument with the value specified at code generation time. If the values do not match, an error occurs.

Value	Description
`false`	The SIL or PIL execution does not compare the value that a test file provides for a constant input argument with the value specified at code generation time. The SIL or PIL execution uses the value specified at code generation time. If the test file uses a different value, then the results in MATLAB might differ from the results in the SIL or PIL execution. It is possible to speed up a SIL or PIL execution by setting `SILPILCheckConstantInputs` to `false`.

See “Speed Up SIL/PIL Execution by Disabling Constant Input Checking and Global Data Synchronization” (Embedded Coder)

`SILPILSyncGlobalData` — Global data synchronization mode for a SIL or PIL execution

`true` (default) | `false`

Global data synchronization mode for a SIL or PIL execution, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. A SIL or PIL execution synchronizes the values of global variables in the SIL or PIL execution environment with the values in the MATLAB workspace. If a global variable is constant and its value in the SIL or PIL execution environment differs from its value in the MATLAB workspace, an error occurs.

Value	Description
`false`	The SIL or PIL execution does not synchronize the values of global variables in the SIL or PIL execution environment with the values in the MATLAB workspace. If the values are not synchronized, the results in MATLAB might differ from the results in the SIL or PIL execution. It is possible to speed up a SIL or PIL execution by setting `SILPILSyncGlobalData` to `false`.

See “Speed Up SIL/PIL Execution by Disabling Constant Input Checking and Global Data Synchronization” (Embedded Coder)

`StackUsageMax` — Maximum stack usage per application
`200000` (default) | positive integer

Maximum stack usage per application, in bytes, specified as a positive integer. Set a limit that is lower than the available stack size. Otherwise, a run-time stack overflow might occur. The C compiler detects and reports stack overflows.

See “Disable Support for Integer Overflow or Nonfinites”.

`SupportNonFinite` — Support for nonfinite values
`true` (default) | `false`

Support for nonfinite values, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If `GenerateNonFiniteFilesIfUsed` is set to `true`, the code generator produces code to support nonfinite values (`Inf` and `NaN`) only if they are used. If `GenerateNonFiniteFilesIfUsed` is set to `false`, the code generator always produces code to support nonfinite values (`Inf` and `NaN`).
`false`	The code generator does not produce code to support nonfinite values.

See “Disable Support for Integer Overflow or Nonfinites”.

`TargetLang` — Language to use in generated code
`'C'` (default) | `'C++'`

Language to use in generated code, specified as `'C'` or `'C++'`. If you specify C++, the code generator wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Dependency: If `DeepLearningConfig` is set, `codegen` sets `TargetLang` to `C++`.

`TargetLangStandard` — Standard math library to use for the generated code
`'C89/C90 (ANSI)'` | `'C99 (ISO)'` | `'C++03 (ISO)'`

Standard math library to use for the generated code, specified as one of these character vectors:

- `'C89/C90 (ANSI)'`
- `'C99 (ISO)'`
- `'C++03 (ISO)'`

The code generator uses the standard math library for calls to math operations. The default standard math library depends on the language that you select. For C, the default library is `'C99 (ISO)'`. For C++, the default library is `'C++03 (ISO)'`.

See “Change the Standard Math Library”.

`Toolchain` — Toolchain to use for building a C/C++ library or executable program
`'Automatically locate an installed toolchain'` (default) | character vector

Toolchain to use for building a C/C++ library or executable program, specified as a character vector. The list of available toolchains depends on the host computer platform, and can include custom toolchains that you added. If you do not specify a toolchain, the code generator locates an installed toolchain.

Note:

- Suppose that you create a `coder.CodeConfig` object `cfg` in a MATLAB session and use it in another MATLAB session. If the MATLAB host computer for the second session does not have the toolchain specified in the `cfg.Toolchain` property installed on it, this parameter reverts to its default value. The default value is `'Automatically locate an installed toolchain'`.

`Verbose` — Code generation progress display
false (default) | `true`

Code generation progress display, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not display code generation progress.
`true`	The code generator displays code generation progress, including code generation stages and compiler output.

`VerificationMode` — Code verification mode
`'None'` (default) | `'SIL'` | `'PIL'`

Code verification mode, specified as one of the values in this table.

Value	Description
`'None'`	Normal execution
`'SIL'`	Software-in-the-loop (SIL) execution
`'PIL'`	Processor-in-the-loop (PIL) execution

See “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” (Embedded Coder).

Examples

Specify Configuration Parameters for Generation of Standalone Code with Embedded Coder

Write a MATLAB function from which you can generate code. This example uses the function `myadd` that returns the sum of its inputs.

```
function c = myadd(a,b)
c = a + b;
end
```

Create a configuration object for generation of standalone C/C++ code (a static library, a dynamically linked library, or an executable program). For example, create a configuration object for generation of a static library.

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

If Embedded Coder is installed, `coder.config` creates a `coder.EmbeddedCodeConfig` object.

Change the values of the properties for which you do not want to use the default values. For example,change the comment style in the generated code to single-line comments preceded by `//`.

```
cfg.CommentStyle = 'Single-line';
```

The `CommentStyle` property is available only in an Embedded Coder configuration object.

Generate code by using `codegen`. Pass the configuration object to `codegen` by using the `-config` option. Specify that the input arguments are scalar double.

```
codegen myadd -config cfg -args {1 1} -report
```

Alternative Functionality

To use default configuration parameters for build types `'LIB'`, `'DLL'`, or `'EXE'`, use the `codegen` option `-config:lib`, `-config:dll`, or `-config:exe`, respectively. Then, you do not have to create and pass a configuration object to `codegen`.

See Also

Functions
`codegen` | `coder.config` | `pathsep`

Objects
`coder.CodeConfig` | `coder.HardwareImplementation` | `coder.MexCodeConfig`

Topics

“Generating Standalone C/C++ Executables from MATLAB Code”
“Generating C/C++ Static Libraries from MATLAB Code”
“Generating C/C++ Dynamically Linked Libraries from MATLAB Code”

Introduced in R2011a

coder.hardware

Create hardware board configuration object for C/C++ code generation from MATLAB code

Description

The `coder.hardware` function creates a `coder.Hardware` object that contains hardware board parameters for C/C++ code generation from MATLAB code.

To use a `coder.Hardware` object for code generation, assign it to the `Hardware` property of a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object that you pass to `codegen`. Assigning a `coder.Hardware` object to the `Hardware` property customizes the associated `coder.HardwareImplementation` object and other configuration parameters for the particular hardware board.

Creation

Syntax

```
coder.hardware(boardname)
coder.hardware()
```

Description

`coder.hardware(boardname)` creates a `coder.Hardware` object for the specified hardware board. The board must be supported by an installed support package. To see a list of available boards, call `coder.hardware` without input parameters.

`coder.hardware()` returns a cell array of names of boards supported by installed support packages.

Input Arguments

`boardname` — hardware board name
character vector | string scalar

Hardware board name, specified as a character vector or a string scalar.

Example: `'Raspberry Pi'`

Example: `"Raspberry Pi"`

Properties

`Name` — Name of hardware board
character vector | string scalar

Name of hardware board, specified as a character vector or a string scalar. The `coder.hardware` function sets this property using the `boardname` argument.

`CPUClockRate` — Clock rate of hardware board
100 (default) | double scalar

Clock rate of hardware board, specified as a double scalar.

Examples

Generate Code for a Supported Hardware Board

Configure code generation for a Raspberry Pi board and generate code for a function `foo`.

```
hwlist = coder.hardware();
if ismember('Raspberry Pi',hwlist)
    hw = coder.hardware('Raspberry Pi');
    cfg = coder.config('lib');
    cfg.Hardware = hw;
    codegen foo -config cfg -report
end
```

Check Supported Hardware Boards

Before creating a `coder.Hardware` object for a hardware board, check that the board is supported by an installed support package.

List all boards for which a support package is installed.

```
hwlist = coder.hardware()
```

Test for an installed support package for a particular board.

```
hwlist = coder.hardware();
if ismember('Raspberry Pi',hwlist)
    hw = coder.hardware('Raspberry Pi');
end
```

Tips

- In addition to the `Name` and `CPUClockRate` properties, a `coder.Hardware` object has dynamic properties specific to the hardware board.
- To configure code generation parameters for processor-in-the-loop (PIL) execution on a supported hardware board, use `coder.hardware`. See “PIL Execution with ARM Cortex-A at the Command Line” (Embedded Coder) and “PIL Execution with ARM Cortex-A by Using the MATLAB Coder App” (Embedded Coder). PIL execution requires Embedded Coder.

See Also

`codegen` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.HardwareImplementation`

Introduced in R2015b

coder.HardwareImplementation

Hardware-specific configuration parameters for C/C++ code generation from MATLAB code

Description

A `coder.HardwareImplementation` object contains hardware-specific configuration parameters that `codegen` uses for generating a static library, a dynamically linked library, or an executable program.

To use a `coder.HardwareImplementation` object, you must associate it with a configuration object (a `coder.CodeConfig` object or a `coder.EmbeddedCodeConfig` object) that you pass to `codegen`. To create a `coder.HardwareImplementation` object and associate it with a configuration object, create the configuration object by using `coder.config`.

Access `coder.HardwareImplementation` properties in one of these ways:

- The `HardwareImplementation` property of the associated configuration object. For example:

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

- A dialog box for the associated configuration object. See “Access Hardware Implementation Properties with a Dialog Box” on page 4-95.

By default, the `coder.HardwareImplementation` properties specify characteristics of the MATLAB host computer. To specify a different device, modify the `ProdHWDeviceType` property. The hardware device determines the values of the other device-specific properties and the properties that you can change.

Creation

To create a `coder.HardwareImplementation` object, use `coder.config` to create a configuration object (a `coder.CodeConfig` object or a `coder.EmbeddedCodeConfig` object) for generation of standalone code. When `coder.config` creates the configuration

object, it also creates a `coder.HardwareImplementation` object. `coder.config` sets the `HardwareImplementation` property of the configuration object to the `coder.HardwareImplementation` object.

Properties

`Description` — Object description
`'class HardwareImplementation: Hardware implementation specifications.'` (default) | character vector

Description of `coder.HardwareImplementation` object, specified as a character vector.

`Name` — Object name
`'HardwareImplementation'` (default) | character vector

Object name, specified as a character vector.

`ProdEqTarget` — Equivalence of production and target hardware characteristics
`true` (default) | `false`

Equivalence of production and target (test) hardware characteristics, specified as `true` or `false`.

A `coder.HardwareImplementation` object has two sets of hardware properties—one for the characteristics of the production hardware and one for the characteristics of the target (test) hardware. By default, `codegen` uses the properties of the production hardware. Typically, you leave `ProdEqTarget` set to `true` and work only with the production properties.

Setting `ProdEqTarget` to `false` is an advanced feature that enables the target hardware properties. If you change `ProdEqTarget` to `false`, `codegen` uses the properties for the target hardware and generates code to emulate the behavior of the production hardware on the target hardware. If you set `ProdEqTarget` to `false` for testing, before you deploy the generated code to the production hardware, set `ProdEqTarget` to `true` and regenerate the code.

At the beginning of an Embedded Coder processor-in-the-loop (PIL) execution, the software checks the hardware implementation properties with reference to the target hardware on which the PIL execution runs. If `ProdEqTarget` is `true`, the software

checks the production properties (properties that start with `Prod`). If `ProdEqTarget` is `false`, the software checks the target properties (properties that start with `Target`). See "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" (Embedded Coder).

Production Hardware Properties

`ProdBitPerChar` — Length in bits of the C `char` data type

8 (default) | multiple of 8, from 8 through 32

Length in bits of the C char data type that the production hardware supports, specified as an integer value from 8 through 32. The value must be a multiple of 8.

`ProdBitPerDouble` — Length in bits of the C `double` data type

64

This property is read-only.

Length in bits of the C double data type that the production hardware supports.

`ProdBitPerFloat` — Length in bits of the C floating-point data type

32

This property is read-only.

Length in bits of the C floating-point data type that the production hardware supports.

`ProdBitPerInt` — Length in bits of the C `int` data type

32 (default) | multiple of 8, from 8 through 32

Length in bits of the C `int` data type that the production hardware supports, specified as an integer value from 8 through 32. The value must be a multiple of 8.

`ProdBitPerLong` — Length in bits of the C `long` data type

32 (default) | multiple of 8, from 32 through 128

Length in bits of the C `long` data type that the production hardware supports, specified as an integer value from 32 through 128. The value must be a multiple of 8.

`ProdBitPerLongLong` — Length in bits of the C `long long` data type

64 (default) | multiple of 8, from 64 through 128

Length in bits of the C `long long` data type that the production hardware supports, specified as an integer value from 64 through 128. The value must be a multiple of 8.

Use the C `long long` data type only if your C compiler supports `long long`.

Dependency:

- `ProdLongLongMode` enables use of the `ProdBitPerLongLong` property.

`ProdBitPerPointer` — Length in bits of pointer data
64 (default) | multiple of 8, from 8 through 64

Length in bits of pointer data that the production hardware supports, specified as an integer value from 8 through 64. The value must be a multiple of 8.

`ProdBitPerPtrDiffT` — Length in bits of `ptrdiff_t` data
64 (default) | 8 | 16 | 24 | 32 | 40 | 64 | 128

Length in bits of `ptrdiff_t` data that the production hardware supports, specified as 8, 16, 24, 32, 40, 64, or 128. The value must be greater than or equal to the value of `ProdBitPerInt`.

`ProdBitPerShort` — Length in bits of the C `short` data type
16 (default) | multiple of 8, from 8 through 32

Length in bits of the C `short` data type that the production hardware supports, specified as a multiple of 8, from 8 through 32.

`ProdBitPerSizeT` — Length in bits of `size_t` data
64 (default) | 8 | 16 | 24 | 32 | 40 | 64 | 128

Length in bits of `size_t` data that the production hardware supports, specified as 8, 16, 24, 32, 40, 64, or 128. The value must be greater than or equal to the value of `ProdBitPerInt`.

`ProdEndianess` — Significance of the first byte of a data word
`'LittleEndian'` (default) | `'BigEndian'` | `'Unspecified'`

Significance of the first byte of a data word for the production hardware, specified as one of the values in this table.

Value	Description
`'LittleEndian'`	This value is the default value. The least significant byte appears first in the byte ordering.
`'BigEndian'`	The most significant byte appears first in the byte ordering.
`'Unspecified'`	The code determines the endianness of the hardware. This value produces less efficient code.

`ProdHWDeviceType` — Manufacturer and type of the production hardware board
`'Generic->MATLAB Host Computer'` (default) | character vector

Manufacturer and type of the production hardware board, specified as a character vector.

Specifying the `ProdHWDeviceType` property sets device-specific properties for the production hardware. You can modify a device-specific property only if it is enabled for the specified hardware.

`codegen` cannot generate code for ASICs or FPGAs. If you set `ProdHWDeviceType` to `'ASIC/FPGA->ASIC/FPGA'`, `ProdEqTarget` becomes `false` so that the code generator uses the properties for the target(test) device.`'ASIC/FPGA->ASIC/FPGA'` is not a valid value for `TargetHWDeviceType`.

`ProdIntDivRoundTo` — Rounding for division of two signed integers
`'Zero'` (default) | `'Undefined'` | `'Floor'`

Rounding by the compiler for the production hardware of the result of division of two signed integers, specified as one of the values in this table.

Value	Description
`'Zero'`	This value is the default value. If the quotient is between two integers, the compiler rounds to the integer that is closer to zero.

Value	Description
`'Floor'`	If the quotient is between two integers, the compiler rounds to the integer that is closer to negative infinity.
`'Undefined'`	The compiler does not round toward zero or negative infinity, or the compiler rounding behavior is undefined.

`ProdLongLongMode` — Long long data type support
`true` (default) | `false`

C compiler support for the long long data type, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator uses the C `long long` data type in the generated code.
`false`	The code generator does not use the C `long long` data type in the generated code.

Most C99 compilers support `long long`.

Tips:

- `ProdLongLongMode` is enabled only if the specified production hardware supports the C `long long` data type.
- If your compiler does not support C `long long`, do not select this parameter.

Dependency:

- `ProdLongLongMode` enables use of the `ProdBitPerLongLong` property.

`ProdShiftRightIntArith` — Implementation of signed integer right shift as arithmetic right shift
true (default) | false

Implementation by the C compiler for the production hardware of a signed integer right shift as an arithmetic right shift, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Indicates that the C compiler implements a right shift of a signed integer as an arithmetic right shift. An arithmetic right shift fills the bit vacated by the right shift with the value of the most significant bit. The most significant bit indicates the sign of the number.
`false`	Indicates that the C compiler does not implement a right shift of a signed integer as an arithmetic right shift.

`ProdWordSize` — Microprocessor native word size
64 (default) | multiple of 8, from 8 through 64

Microprocessor native word size for the production hardware, specified as an integer value from 8 through 64. The value must be a multiple of 8.

Target Hardware Properties

`TargetBitPerChar` — Length in bits of the C `char` data type
8 (default) | multiple of 8, from 8 through 32

Length in bits of the C char data type that the target hardware supports, specified as an integer value from 8 through 32. The value must be a multiple of 8.

`TargetBitPerDouble` — Length in bits of the C `double` data type
64

This property is read-only.

Length in bits of the C double data type that the target hardware supports.

`TargetBitPerFloat` — Length in bits of the C floating-point data type
32

This property is read-only.

Length in bits of the C floating-point data type that the target hardware supports.

`TargetBitPerInt` — Length in bits of the C `int` data type
32 (default) | multiple of 8, from 8 through 32

Length in bits of the C `int` data type that the target hardware supports, specified as an integer value from 8 through 32. The value must be a multiple of 8.

`TargetBitPerLong` — Length in bits of the C `long` data type
32 (default) | multiple of 8, from 32 through 128

Length in bits of the C `long` data type that the target hardware supports, specified as an integer value from 32 through 128. The value must be a multiple of 8.

`TargetBitPerLongLong` — Length in bits of the C `long long` data type
64 (default) | multiple of 8, from 64 through 128

Length in bits of the C `long long` data type that the target hardware supports, specified as an integer value from 64 through 128. The value must be a multiple of 8.

Use the C `long long` data type only if your C compiler supports `long long`.

Dependency:

- `TargetLongLongMode` enables use of the `TargetBitPerLongLong` property.

`TargetBitPerPointer` — Length in bits of pointer data
64 (default) | multiple of 8, from 8 through 64

Length in bits of pointer data that the target hardware supports, specified as an integer value from 8 through 64. The value must be a multiple of 8.

`TargetBitPerPtrDiffT` — Length in bits of `ptrdiff_t` data
64 (default) | 8 | 16 | 24 | 32 | 40 | 64 | 128

Length in bits of `ptrdiff_t` data, specified as 8, 16, 24, 32, 40, 64, or 128. The value must be greater than or equal to the value of `ProdBitPerInt`.

`TargetBitPerShort` — Length in bits of the C `short` data type
16 (default) | multiple of 8, from 8 through 32

Length in bits of the C `short` data type that the target hardware supports, specified as a multiple of 8, from 8 through 32.

`TargetBitPerSizeT` — Length in bits of `size_t` data
64 (default) | 8 | 16 | 24 | 32 | 40 | 64 | 128

Length in bits of `size_t` data that the target hardware supports, specified as 8, 16, 24, 32, 40, 64, or 128. The value must be greater than or equal to the value of `ProdBitPerInt`.

`TargetEndianess` — Significance of the first byte of a data word
`'LittleEndian'` (default) | `'BigEndian'` | `'Unspecified'`

Significance of the first byte of a data word for the target hardware, specified as one of the values in this table.

Value	Description
`'LittleEndian'`	This value is the default value. The least significant byte appears first in the byte ordering.
`'BigEndian'`	The most significant byte appears first in the byte ordering.
`'Unspecified'`	The code determines the endianness of the hardware. This value produces less efficient code.

`TargetHWDeviceType:` — Manufacturer and type of the target (test) hardware board
`'Generic->MATLAB Host Computer'` (default) | character vector

Manufacturer and type of the target (test) hardware board, specified as a character vector.

Specifying the `TargetHWDeviceType` property sets values for the device-specific properties for the target hardware. You can modify a device-specific property only if it is enabled for the specified hardware.

`codegen` cannot generate code for ASICs or FPGAs. If you set `ProdHWDeviceType` to `'ASIC/FPGA->ASIC/FPGA'`, `ProdEqTarget` becomes `false` so that the code generator uses the properties for the target(test) device. `'ASIC/FPGA->ASIC/FPGA'` is not a valid value for `TargetHWDeviceType`

`TargetIntDivRoundTo` — Rounding for division of two signed integers
`'Zero'` (default) | `'Undefined'` | `'Floor'`

Rounding by the compiler for the test hardware of the result of division of two signed integers, specified as one of the values in this table.

Value	Description
`'Zero'`	This value is the default value. If the quotient is between two integers, the compiler rounds to the integer that is closer to zero.
`'Floor'`	If the quotient is between two integers, the compiler rounds to the integer that is closer to negative infinity.
`'Undefined'`	The compiler does not round toward zero or negative infinity, or the compiler rounding behavior is undefined.

`TargetLongLongMode` — Long long data type support
`true` (default) | `false`

C compiler support for the long long data type, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator uses the C `long long` data type in the generated code.
`false`	The code generator does not use the C `long long` data type in the generated code.

Most C99 compilers support `long long`.

Tips:

- `TargetLongLongMode` is enabled only if the specified production hardware supports the C `long long` data type.

- If your compiler does not support C `long long`, do not select this parameter.

Dependency:

- `TargetLongLongMode` enables use of the `TargetBitPerLongLong` property.

`TargetShiftRightIntArith` — Implementation of signed integer right shift as arithmetic right shift
true (default) | false

Implementation by the C compiler for the production hardware of a signed integer right shift as an arithmetic right shift, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Indicates that the C compiler implements a right shift of a signed integer as an arithmetic right shift. An arithmetic right shift fills the bit vacated by the right shift with the value of the most significant bit. The most significant bit indicates the sign of the number.
`false`	Indicates that the C compiler does not implement a right shift of a signed integer as an arithmetic right shift.

`TargetWordSize` — Microprocessor native word size
64 (default) | multiple of 8, from 8 through 64

Microprocessor native word size for the production hardware, specified as an integer value from 8 through 64. The value must be a multiple of 8.

Examples

Specify Hardware-Specific Parameters for C Code Generation

Create a configuration object for generation of standalone code. For example, create a configuration object for generation of a static library.

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

`coder.config` sets the `HardwareImplementation` property of the configuration object to a `coder.HardwareImplementation` object with default parameter values for the MATLAB host computer.

To specify a different hardware device and customize the device-specific properties, set the ProdHWDeviceType property of the `coder.HardwareImplementation` object to one of the available devices. For example:

```
cfg.HardwareImplementation.ProdHWDeviceType = 'AMD->Athlon 64'
```

To use the `coder.HardwareImplementation` object for code generation, use the `-config` option to specify the code generation configuration object associated with the `coder.HardwareImplementation` object.

```
codegen -config cfg myFunction
```

Access Hardware Implementation Properties with a Dialog Box

Open the dialog box for the configuration object that refers to the `coder.HardwareImplementation` object. For example:

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

In the dialog box, click the **Hardware** tab.

If you installed a support package for your hardware board (requires Embedded Coder), select the board from the options in **Hardware Board**. Otherwise, set **Hardware Board** to `None - select device below`. Then, set the **Device vendor** and **Device type**.

The hardware implementation settings have values for the specified device. To see or modify these settings, click the **Customize hardware implementation** link. You can modify a setting only if it is enabled for the specified device.

Tips

To set the `ProdHWDeviceType` property, select the device from the available options by opening a dialog box for the configuration object that refers to the

`coder.HardwareImplementation` object. See “Access Hardware Implementation Properties with a Dialog Box” on page 4-95.

Alternative Functionality

- You can directly create a `coder.HardwareImplementation` object and assign it to a `coder.CodeConfig` object or a `coder.EmbeddedCodeConfig` object.

  ```
  hw_cfg = coder.HardwareImplementation;
  hw_cfg.ProdHWDeviceType = 'AMD->Athlon 64';
  cfg = coder.config('lib');
  cfg.HardwareImplementation = hw_cfg;
  ```

- If you install a support package for your hardware, you can customize hardware-specific code generation parameters by setting the `Hardware` property of a `coder.EmbeddedCodeConfig` object to a `coder.Hardware` object. To create a `coder.Hardware` object, use `coder.hardware`. For example:

  ```
  cfg = coder.config('lib');
  hw = coder.hardware('Raspberry Pi');
  cfg.Hardware = hw;
  ```

See Also

Functions
`codegen` | `coder.config`

Objects
`coder.CodeConfig` | `coder.EmbeddedCodeConfig`

Topics
“Configure Build Settings”

Introduced in R2011a

coder.MexCodeConfig

Configuration parameters for MEX function generation from MATLAB code

Description

A `coder.MexCodeConfig` object contains the configuration parameters that `codegen` uses when generating a MEX function. Pass the object to the `codegen` function by using the `-config` option.

Creation

Syntax

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

Description

`cfg = coder.config('mex')` creates a `coder.MexCodeConfig` object.

`cfg = coder.config` creates a `coder.MexCodeConfig` object.

Properties

`CompileTimeRecursionLimit` — Maximum number of function specializations for compile-time recursion
50 (default) | positive integer

Maximum number of function specializations for compile-time recursion, specified as a positive integer. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to 0. The default compile-time recursion limit is large enough for most recursive functions that require this type of recursion. If code generation

fails because of the compile-time recursion limit, and you want compile-time recursion, try to increase the limit. Alternatively, change your MATLAB code so that the code generator uses run-time recursion. See “Compile-Time Recursion Limit Reached”.

`ConstantFoldingTimeout` — Maximum number of instructions to be executed by the constant folder
40000 (default) | positive integer

Maximum number of instructions that the constant folder executes. In some situations, code generation requires specific instructions to be constant. If constant folding stops before these instructions are constant-folded, code generation fails. In this case, increase the value of `ConstantFoldingTimeout`.

See “MATLAB Coder Optimizations in Generated Code”.

`ConstantInputs` — Constant input checking mode
`'CheckValues'` (default) | `'IgnoreValues'` | `'Remove'`

Constant input checking mode, specified as one of the values in this table.

Value	Description
`'CheckValues'`	This value is the default value. When you call the MEX function, it checks that the value you provide for a constant input argument is the value specified at code generation time. You can call the MEX function and the original MATLAB function with the same arguments. Therefore, you can use the same test file for both functions. Checking the values can slow down execution of the MEX function.

Value	Description
`'IgnoreValues'`	When you call the MEX function, it ignores the value that you provide for a constant input argument. It uses the value specified at code generation time. You can use the same test file without the overhead of checking the constant argument values.
`'Remove'`	The code generator removes constant input arguments from the MEX function signature. When you call the MEX function, you do not provide a value for a constant input argument. This option provides backward compatibility.

See “Constant Input Checking in MEX Functions”.

`CustomHeaderCode` — Custom code that appears at top of generated C/C++ header files

`' '` (default) | character vector

Custom code that appears near the top of each C/C++ header file generated from your MATLAB code, specified as a character vector.

`CustomInclude` — Include folders to add to include path for compiling generated code

`' '` (default) | character vector

Include folders to add to the include path when compiling the generated code. Specify the list of include folders as a character vector. In the character vector, separate include folders by a `pathsep` character. For example:

```
cfg = coder.config('mex');
cfg.CustomInclude = ['C:\Project' pathsep 'C:\Custom Files'];
```

`CustomInitializer` — Custom code to include in the generated initialize function

`' '` (default) | character vector

Custom code to include in the generated initialize function, specified as a character vector.

`CustomLibrary` — Static library files to link with the generated code
`''` (default) | character vector

Static library files to link with the generated code, specified as a character vector. In the character vector, separate library file names by a `pathsep` character.

`CustomSource` — Source files to compile and link with the generated code
`''` (default) | character vector

Source files to compile and link with the generated code, specified as a character vector. In the character vector, separate source file names by a `pathsep` character.

The build process searches for the source files first in the current folder, and then in the include folders that you specify in `CustomInclude`. If source files with the same name occur in multiple folders on the search path, the build process might use a different file than the file that you specified.

Suppose that you specify `foo.cpp` as a source file. If `foo.c` and `foo.cpp` are both on the search path, you cannot be sure whether the build process uses `foo.c` or `foo.cpp`.

`CustomSourceCode` — Code to appear near the top of the generated `.c` or `.cpp` file
`''` (default) | character vector

Specify code to appear near the top of the generated `.c` or `.cpp` file, outside of any function. Specify code as a character vector.

Do not specify a C static function definition.

`CustomTerminator` — Code that appears in the generated terminate function
`''` (default) | character vector

Code that appears in the generated terminate function, specified as a character vector.

`DeepLearningConfig` — Configuration object for deep learning code generation
`coder.MklDNNConfig` object

Configuration object for code generation for deep learning networks, specified as a `coder.MklDNNConfig` object.

A `coder.MklDNNConfig` object contains parameters specific to C++ code generation for deep learning using Intel MKL-DNN. To create a `coder.MklDNNConfig` object, use `coder.DeepLearningConfig`. For example:

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Code generation for deep learning networks requires Deep Learning Toolbox and the MATLAB Coder Interface for Deep Learning Libraries support package.

See “Code Generation for Deep Learning Networks with MKL-DNN”.

Note MEX code generation is not supported for deep learning using the ARM Compute Library.

Dependency: If `DeepLearningConfig` is set, `codegen` sets `TargetLang` to `C++`.

`DynamicMemoryAllocation` — Dynamic memory allocation mode
`'Threshold'` (default) | `'AllVariableSizeArrays'` | `'Off'`

Dynamic memory allocation mode, specified as one of the values in this table.

Value	Description
`'Threshold'`	This value is the default value. The code generator allocates memory dynamically on the heap for variable-size arrays whose size (in bytes) is greater than or equal to `DynamicMemoryAllocationThreshold`.
`'AllVariableSizeArrays'`	The code generator dynamically allocates memory for all variable-size arrays on the heap.
`'Off'`	The code generator statically allocates memory for variable-size arrays on the stack.

Unbounded variable-size arrays require dynamic memory allocation.

Dependencies:

- `EnableVariableSizing` enables this parameter.
- Setting this `DynamicMemoryAllocation` to `'Threshold'` enables the `DynamicMemoryAllocationThreshold` parameter.

See “Generate Code for Variable-Size Data”.

`DynamicMemoryAllocationThreshold` — Size threshold for dynamic memory allocation of variable-size arrays
`65536` (default) | positive integer

Size threshold for dynamic memory allocation of variable-size arrays, specified as a positive integer. The code generator uses dynamic memory allocation for variable-size arrays whose size (in bytes) is greater than or equal to the threshold.

Dependency:

- Setting `DynamicMemoryAllocation` to `'Threshold'` enables this parameter.

See “Generate Code for Variable-Size Data”.

`EchoExpressions` — Expression echoing
`true` (default) | `false`

Expression echoing, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The MEX function displays the output of statements that do not end with a semicolon.
`false`	The MEX function does not display the output of statements that do not end with a semicolon.

This property does not apply to common visualization functions, such as `disp`, `plot`, or `figure` when they are called as an extrinsic function. MEX functions display the output of those functions regardless of the presence of a semicolon or the value of `EchoExpressions`.

`EnableAutoExtrinsicCalls` — Automatic extrinsic function calls

`true` (default) | `false`

Automatic extrinsic function calls, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator treats some common visualization functions as extrinsic functions. You do not have to declare these functions as extrinsic by using `coder.extrinsic`. This capability reduces the amount of time that you spend making your code suitable for code generation.
`false`	The code generator does not treat common visualization functions as extrinsic functions unless you declare them as extrinsic by using `coder.extrinsic`.

Some common visualization functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions”.

`EnableDebugging` — C compiler debugging mode

`false` (default) | `true`

C compiler debugging mode, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not enable the C compiler debugging mode.
`true`	The code generator enables the C compiler debugging mode. When debugging mode is enabled, the C compiler does not optimize the code. The compilation is faster, but the execution is slower.

`EnableJIT` — Just-in-time (JIT) compilation mode

`false` (default) | `true`

Just-in-time (JIT) compilation mode, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator creates a C/C++ MEX function by generating and compiling C/C++ code.
`true`	The code generator uses just-in-time (JIT) compilation technology for MEX function generation. The code generator creates a JIT MEX function that contains an abstract representation of the MATLAB code. When you run the JIT MEX function, MATLAB generates the executable code in memory.

To speed up generation of MEX functions, set `EnableJIT` to `true`.

JIT compilation is incompatible with certain code generation features and options, such as custom code or use of the OpenMP library. If you specify JIT compilation and the code generator is unable to use it, it generates a C/C++ MEX function with a warning. If `EnableJIT` and `EnableOpenMP` are `true`, and your code uses `parfor`, the code generator uses JIT compilation and treats the `parfor`-loops as `for`-loops.

See “Speed Up MEX Generation by Using JIT Compilation”.

`EnableMemcpy` — `memcpy` optimization

`true` (default) | `false`

`memcpy` optimization, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the `memcpy` optimization. To optimize code that copies consecutive array elements, the `memcpy` optimization replaces the code with a `memcpy` call. When the number of elements to copy is known at compile time, the code generator uses the `MemcpyThreshold` property to determine whether to use the optimization. See “memcpy Optimization”.
`false`	The code generator does not use the `memcpy` optimization.

`EnableMexProfiling` — Instrumentation for profiling
`false` (default) | `true`

Enabling profiling of generated MEX function, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not include the instrumentation for profiling in the generated MEX function.
`true`	The code generator includes the instrumentation for profiling in the generated MEX function. You can then use the MATLAB Profiler to profile the MEX. See “Profile MEX Functions by Using MATLAB Profiler”.

`EnableOpenMP` — Parallelization of `parfor`-loops
`true` (default) | `false`

Parallelization of `parfor`-loops, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the OpenMP library to produce loop iterations that run in parallel.
`false`	The code generator treats `parfor`-loops as `for`-loops.

See `parfor`.

Use of the OpenMP library is not compatible with just-in-time (JIT) compilation. If `EnableJIT` and `EnableOpenMP` are `true`, the code generator uses JIT compilation and treats `parfor`-loops as `for`-loops.

`EnableRuntimeRecursion` — Run-time recursion support
`true` (default) | `false`

Run-time recursion support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Recursive functions are allowed in the generated code.
`false`	Recursive functions are not allowed in the generated code.

Some coding standards, such as MISRA, do not allow recursion. To increase the likelihood of generating code that is compliant with MISRA C, set `EnableRuntimeRecursion` to `false`.

If your MATLAB code requires run-time recursion and `EnableRuntimeRecursion` is `false`, code generation fails.

See “Code Generation for Recursive Functions”.

`EnableVariableSizing` — Variable-size array support
`true` (default) | `false`

Variable-size array support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Variable-size arrays are allowed for code generation.
`false`	Variable-size arrays are not allowed for code generation.

Dependency:

- Enables `Dynamic memory allocation`.

See “Code Generation for Variable-Size Arrays”.

`ExtrinsicCalls` — Extrinsic function call support
`true` (default) | `false`

Extrinsic function call support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. For an extrinsic function, the code generator produces a call to a MATLAB function. The code generator does not generate the internal code for the function.

Value	Description
`false`	The code generator ignores an extrinsic function. It does not generate code for the call to the MATLAB function. If the extrinsic function affects the output of the MATLAB function, the code generator issues a compilation error. If you set `ExtrinsicCalls` to `false`, the generated MEX function cannot display run-time messages from `error` or `assert` statements in your MATLAB code. The MEX function reports that it cannot display the error message. To see the error message, set `ExtrinsicCalls` to `true` and generate the MEX function again.

The value of `ExtrinsicCalls` affects how a MEX function generates random numbers for rand, `randi`, and `randn`. If `ExtrinsicCalls` is `true`, the MEX function uses the MATLAB global random number stream to generate random numbers. Otherwise, the MEX function uses a self-contained random number generator.

See “Extrinsic Functions”.

`FilePartitionMethod` — File partitioning mode
`'MapMFileToCFile'` (default) | `'SingleFile'`

File partitioning mode specified as one of the values in this table.

Value	Description
`'MapMFileToCFile'`	This value is the default value. The code generator produces separate C/C++ files for each MATLAB language file.
`'SingleFile'`	The code generator produces a single file for C/C++ functions that map to your MATLAB entry-point functions. The code generator produces separate C/C++ files for utility functions.

See “How MATLAB Coder Partitions Generated Code”.

`GenCodeOnly` — Generation of only source code

`false` (default) | `true`

Generation of only source code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator produces C/C++ source code and builds object code.
`true`	The code generator produces C/C++ source code, but does not invoke the make command or build object code. When you iterate between modifying MATLAB code and generating C/C++ code, generating only code can save time.

`GenerateComments` — Comments in generated code

`true` (default) | `false`

Comments in generated code, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator places comments in the generated code.
`false`	The code generator does not place comments in the generated code.

`GenerateReport` — Code generation report

`false` (default) | true

Code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator produces a report only if error or warning messages occur, or if you set `LaunchReport` to `true`.
`true`	The code generator produces a code generation report.

`GlobalDataSyncMethod` — Global data synchronization mode
`'SyncAlways'` (default) | `'SyncAtEntryAndExits'` | `'NoSync'`

Global data synchronization mode, specified as one of the values in this table.

Value	Description for Global Data	Description for Constant Global Data
`'SyncAlways'` (default)	This value is the default value. Synchronizes global data at MEX function entry and exit and for extrinsic calls for maximum consistency between MATLAB and the generated MEX function. To maximize performance, if the extrinsic calls do not change global data, use this option with the `coder.extrinsic -sync:off` option to turn off synchronization for these calls.	Verifies consistency of constant global data at MEX function entry and after extrinsic calls. If the global data values in the MATLAB global workspace are inconsistent with the compile-time constant global values in the MEX function, the MEX function ends with an error. Use the `coder.extrinsic -sync:off` option to turn off consistency checks after specific extrinsic calls.

Value	Description for Global Data	Description for Constant Global Data
`'SyncAtEntryAndExits'`	Synchronizes global data at MEX function entry and exit only. To maximize performance, if only a few extrinsic calls change global data, use this option with the `coder.extrinsic -sync:on` option to turn on synchronization for these calls.	Verifies constant global data at MEX function entry only. If the global data values in the MATLAB global workspace are inconsistent with the compile-time constant global values in the MEX function, the MEX function ends with an error. Use the `coder.extrinsic -sync:on` option to turn on consistency checks after specific extrinsic calls.
`'NoSync'`	Disables synchronization. Before disabling synchronization, verify that your MEX function does not interact with MATLAB global data. Otherwise, inconsistencies between MATLAB and the MEX function can occur.	Disables consistency checks.

See “Generate Code for Global Data”.

`HighlightPotentialRowMajorIssues` — Potential row-major layout issues
`true` (default) | `false`

Display of potential row-major layout efficiency issues, specified as one of the values in this table.

Value	Description
`true`	The code generation report displays potential efficiency issues due to row-major layout. (This value is the default value.)
`false`	The code generation report does not display issues related to array layout.

See “Code Design for Row-Major Array Layout”.

`InitFltsAndDblsToZero` — Assignment of float and double zero with memset
`true` (default) | `false`

Assignment of float and double zero with `memset`, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. If possible, the code generator uses the `memset` optimization for assignment of floating-point zero to consecutive array elements. To assign consecutive array elements, the `memset` optimization uses a `memset` call. When the number of elements to assign is known at compile time, the code generator uses the `MemcpyThreshold` property to determine whether to use the optimization. See “memset Optimization”.
`false`	The code generator does not use the `memset` optimization for assignment of float and double zero to consecutive array elements.

`InlineStackLimit` — Stack size limit for inlined functions
4000 (default) | positive integer

Stack size limit for inlined functions, specified as a positive integer. The stack size limit determines the amount of stack space allocated for local variables of the inlined function.

Specifying a limit for the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even when the function returns.

This capability is especially important for embedded processors where stack size can be limited.

See “Control Inlining”.

`InlineThreshold` — Function size threshold for inlining

10 (default) | positive integer

Function size threshold for inlining, specified as a positive integer.

Unless other conditions prevent inlining, the code generator inlines functions that are smaller than the threshold.

The function size is measured as an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. To obtain the inlining behavior that you want, experiment with the threshold. For example, if the default threshold results in inlining of large functions and generation of large amounts of C code, tune the threshold until you are satisfied with the size of the generated code.

See “Control Inlining”.

`InlineThresholdMax` — Maximum size of functions after inlining

200 (default) | positive integer

Maximum size of functions after inlining, specified as a positive integer. If, after inlining, the size of the calling function exceeds `InlineThresholdMax`, then the code generator does not inline the called function.

The function size is measured as an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. To obtain the inlining behavior that you want, experiment with the threshold. For example, if the default threshold results in inlining of large functions and generation of large amounts of C code, tune the threshold until you are satisfied with the size of the generated code.

See “Control Inlining”.

`IntegrityChecks` — Memory integrity checking

`true` (default) | false

Memory integrity checking, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The generated code detects memory integrity violations and stops execution with a diagnostic message.
`false`	The generated code does not detect memory integrity violations. Setting `IntegrityChecks` to `false` can improve performance. However, without memory integrity checks, violations result in unpredictable behavior. Set `IntegrityChecks` to `false` only if you have verified that array bounds checking and dimension checking are unnecessary. Setting `IntegrityChecks` to `false` also disables the run-time stack.

See “Control Run-Time Checks”.

`LaunchReport` — Automatic open of code generation report
`false` (default) | true

Automatic open of code generation report, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. If errors or warnings occur, or if `GenerateReport` is `true`, the code generator produces a report, but does not open the report.
`true`	The code generator produces and opens a code generation report.

`MATLABSourceComments` — Inclusion of MATLAB source code as comments in generated code
`false` (default) | `true`

Inclusion of MATLAB source code as comments in generated code, specified as one of the values in this table.

Value	Description
`false`	This value is the default value. The code generator does not insert MATLAB source code as comments in the generated code. The code generator does not include the MATLAB function signature in the function banner.
`true`	The code generator inserts MATLAB source code as comments in the generated code. A traceability tag immediately precedes each line of source code. The traceability tag helps you to locate the corresponding MATLAB source code. See “Tracing Generated C/C++ Code to MATLAB Source Code”. The code generator also includes the MATLAB function signature in the function banner.

Dependency:

- `GenerateComments` enables this parameter.

See “Tracing Generated C/C++ Code to MATLAB Source Code”.

`MemcpyThreshold` — Minimum size for `memcpy` or `memset` optimization
64 (default) | positive integer

Minimum size, in bytes, for memcpy or memset optimization, specified as a positive integer.

To optimize generated code that copies consecutive array elements, the code generator tries to replace the code with a `memcpy` call. To optimize generated code that assigns a literal constant to consecutive array elements, the code generator tries to replace the code with a `memset` call.

The number of bytes is the number of array elements to copy or assign multiplied by the number of bytes required for the C/C++ data type.

If the number of elements to copy or assign is variable (not known at compile time), the code generator ignores the `MemcpyThreshold` property.

See “memcpy Optimization” and “memset Optimization”.

`Name` — Object name
`'MexCodeConfig'` (default) | character vector

Object name, specified as a character vector.

`PostCodeGenCommand` — Command to customize build processing
`''` (default) | character vector

Command to customize build processing after MEX function generation with `codegen`, specified as a character vector.

See “Build Process Customization”.

`PreserveArrayDimensions` — N-dimensional indexing
`false` (default) | `true`

Generation of code that uses N-dimensional indexing, specified as one of the values in this table.

Value	Description
`false`	Generate code that uses one-dimensional indexing. (This value is the default value.)
`true`	Generate code that uses N-dimensional indexing.

See “Generate Code That Uses N-Dimensional Indexing”.

`PreserveVariableNames` — Variable names to preserve in the generated code
`'None'` (default) | `'UserNames'` | `'All'`

Variable names to preserve in the generated code, specified as one of the values in this table.

Value	Description
`'None'`	This value is the default value. The code generator does not have to preserve any variable names. It can reuse any variables that meet the requirements for variable reuse. If your code uses large structures or arrays, setting `PreserveVariableNames` to `'None'` can reduce memory usage or improve execution speed.
`'UserNames'`	The code generator preserves names that correspond to variables that you define in the MATLAB code. It does not replace your variable name with another name and does not use your name for another variable. To improve readability, set `PreserveVariableNames` to `'UserNames'`. Then, you can more easily trace the variables in the generated code back to the variables in your MATLAB code. Setting `PreserveVariableNames` to `'UserNames'` does not prevent an optimization from removing your variables from the generated code or prevent the C/C++ compiler from reusing the variables in the generated binary code.
`'All'`	Preserve all variable names. This parameter value disables variable reuse. Use it only for testing or debugging, not for production code.

See “Preserve Variable Names in Generated Code”.

`ReportInfoVarName` — Name of variable containing code generation report information

`''` (default) | character vector

Name of variable to which you export information about code generation, specified as a character vector. The code generator creates this variable in the base MATLAB workspace. This variable contains information about code generation settings, input files, generated files, and code generation messages.

See “Access Code Generation Report Information Programmatically” and coder.ReportInfo Properties.

`ReportPotentialDifferences` — Potential differences reporting
`true` (default) | `false`

Potential difference reporting, specified as one of the values in this table:

Value	Description
`true`	The code generator reports potential behavior differences between generated code and MATLAB code. The potential differences are listed on a tab of the code generation report. A potential difference is a difference that occurs at run time only under certain conditions.
`false`	The code generator does not report potential differences.

See “Potential Differences Reporting”.

`ReservedNameArray` — Names that the code generator cannot use for functions or variables
`''` (default) | character vector

Names that code generator cannot use for functions or variables, specified as a character vector.

`ResponsivenessChecks` — Responsiveness checks
`true` (default) | `false`

Responsiveness checks, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. You can use **Ctrl+C** to stop execution of a generated MEX function.
`false`	To end a long-running MEX function, you might have to terminate MATLAB.

See “Control Run-Time Checks”.

`RowMajor` — Row-major array layout
`false` (default) | `true`

Generation of code that uses row-major array layout, specified as one of the values in this table.

Value	Description
`false`	Generate code that uses column-major array layout. (This value is the default value.)
`true`	Generate code that uses row-major array layout.

See “Generate Code That Uses Row-Major Array Layout”.

`SaturateOnIntegerOverflow` — Integer overflow support
`true` (default) | `false`

Integer overflow support, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator produces code to handle integer overflow. Overflows saturate to either the minimum or maximum value that the data type can represent.

Value	Description
`false`	The code generator does not produce code to handle integer overflow. Do not set `SaturateOnIntegerOverflow` to `false` unless you are sure that your code does not depend on integer overflow support. If you disable integer overflow support and integrity checks are enabled, the generated code produces an error for overflows. If you disable integer overflow support and you disable integrity checks, the overflow behavior depends on your target C compiler. In the C standard, the behavior for integer overflow is undefined. However, most C compilers wrap on overflow.

This parameter applies only to MATLAB built-in integer types. It does not apply to doubles, singles, or fixed-point data types.

See “Disable Support for Integer Overflow or Nonfinites”.

`StackUsageMax` — Maximum stack usage per application
`200000` (default) | positive integer

Maximum stack usage per application, in bytes, specified as a positive integer. Set a limit that is lower than the available stack size. Otherwise, a run-time stack overflow might occur. The C compiler detects and reports stack overflows.

See “Control Stack Space Usage”.

`TargetLang` — Language to use in generated code
`'C'` (default) | `'C++'`

Language to use in generated code, specified as `'C'` or `'C++'`. If you specify C++, the code generator wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Dependency: If `DeepLearningConfig` is set, `codegen` sets `TargetLang` to `C++`.

Examples

Specify Configuration Parameters for MEX Function Generation

Write a MATLAB function from which you can generate code. This example uses the function `myadd` that returns the sum of its inputs.

```
function c = myadd(a,b)
c = a + b;
end
```

Create a configuration object for MEX function generation.

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

Change the values of the properties for which you do not want to use the default values. For example, enable just-in-time (JIT) compilation.

```
cfg.EnableJIT = true;
```

Generate code by using `codegen`. Pass the configuration object to `codegen` by using the `-config` option. Specify that the input arguments are scalar double.

```
codegen myadd -config cfg -args {1 1} -report
```

Alternative Functionality

To use default configuration parameter values for MEX function generation, instead of creating a configuration object, you can call `codegen` without specifying a configuration object or with the `-config:mex` option.

See Also

Functions
`codegen` | `coder.config` | `pathsep`

Objects
`coder.CodeConfig` | `coder.EmbeddedCodeConfig`

Topics

“Accelerate MATLAB Algorithm by Generating MEX Function”

Introduced in R2011a

coder.ARMNEONConfig

Parameters to configure deep learning code generation with the ARM Compute Library

Description

The `coder.ARMNEONConfig` object contains ARM Compute Library and target specific parameters that `codegen` uses for generating C++ code for deep neural networks.

To use a `coder.ARMNEONConfig` object for code generation, assign it to the `DeepLearningConfig` property of a code generation configuration object that you pass to `codegen`.

Creation

Syntax

```
deepLearningCfg = coder.DeepLearningConfig('arm-compute')
```

Description

`deepLearningCfg = coder.DeepLearningConfig('arm-compute')` creates a `coder.ARMNEONConfig` object for deep learning code generation by using the ARM Compute Library.

Properties

`ArmComputeVersion` — Version of ARM Compute Library
'18.05' (default) | '18.03'

Version of ARM Compute Library used on the target hardware, specified as a character vector or string scalar. If you set `ArmComputeVersion` to a version later than `'18.05'`, `ArmComputeVersion` is set to `'18.05'`.

`ArmArchitecture` — ARM architecture supported in the target hardware
'armv8' | 'armv7'

ARM architecture supported in the target hardware, specified as a character vector or string scalar. The specified architecture must be the same as the architecture for the ARM Compute Library on the target hardware.

`ARMArchitecture` must be specified for these cases:

- You do not use a hardware support package (the `Hardware` property of the code generation configuration object is empty).
- You use a hardware support package, but generate code only.

`TargetLib` — Target library name
'arm-compute'

Name of target library, specified as a character vector.

Examples

Specify Configuration Parameters for C++ Code Generation for the SqueezeNet Network

Create an entry-point function `squeezenet` that uses the `coder.loadDeepLearningNetwork` function to load the `squeezenet` object.

```
function out = squeezenet_predict(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('squeezenet', 'squeezenet');
end

out = predict(mynet,in);
```

Create a `coder.config` configuration object for generation of a static library.

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

Set the target language to C++. Specify that you want to generate only source code.

```
cfg.TargetLang = 'C++';
cfg.GenCodeOnly=true;
```

Create a `coder.ARMNEONConfig` deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv8';
dlcfg.ArmComputeVersion = '18.05';
cfg.DeepLearningConfig = dlcfg;
```

Use the `-config` option of the `codegen` function to specify the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
codegen -args {ones(227,227,3,'single')} -config cfg squeezenet_predict
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the C++ code for the entry-point function `squeezenet_predict.cpp`, header, and source files containing the C++ class definitions for the convolutional neural network (CNN), weight, and bias files.

See Also

`codegen` | `coder.CodeConfig` | `coder.DeepLearningConfig` | `coder.MklDNNConfig`

Topics

“Code Generation for Deep Learning Networks with ARM Compute Library”

Introduced in R2019a

coder.MklDNNConfig

Parameters to configure deep learning code generation with the Intel Math Kernel Library for Deep Neural Networks

Description

The `coder.MklDNNConfig` object contains the Intel MKL-DNN specific parameters that `codegen` uses for generating C++ code for deep neural networks.

To use a `coder.MklDNNConfig` object for code generation, assign it to the `DeepLearningConfig` property of a code generation configuration object that you pass to `codegen`.

Creation

Syntax

```
deepLearningCfg = coder.DeepLearningConfig('mkldnn')
```

Description

`deepLearningCfg = coder.DeepLearningConfig('mkldnn')` creates a `coder.MklDNNConfig` object for deep learning code generation by using the MKL-DNN library.

Properties

`TargetLib` — Target library name
'mkldnn'

Name of target library, specified as a character vector.

Examples

Specify Configuration Parameters for MEX Function Generation for the AlexNet Network

Create an entry-point function `alexneteg` that uses the `coder.loadDeepLearningNetwork` function to load the `alexnet` `SeriesNetwork` object.

```
function out = alexneteg(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('alexnet', 'myalexnet');
end

out = predict(mynet,in);
```

Create a `coder.config` configuration object for MEX code generation.

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

Set the target language to C++.

```
cfg.TargetLang = 'C++';
```

Create a `coder.MklDNNConfig` deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Use the `-config` option of the `codegen` function to pass the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
codegen -args {ones(227,227,3,'single')} -config cfg alexneteg
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the C++ code for the entry-point function `alexneteg.cpp`, header, and source

files containing the C++ class definitions for the convolutional neural network (CNN), weight, and bias files.

See Also

`codegen` | `coder.ARMNEONConfig` | `coder.CodeConfig` | `coder.DeepLearningConfig`

Topics

“Code Generation for Deep Learning Networks with MKL-DNN”
“Code Generation for Deep Learning Networks with ARM Compute Library”
“Code Generation for Deep Learning Networks with cuDNN” (GPU Coder)
“Code Generation for Deep Learning Networks with TensorRT” (GPU Coder)

Introduced in R2018b

coder.OutputType

Output type from an entry-point function to specify as an input type

Description

A `coder.OutputType` object represents the type of an entry-point function output variable. Use `coder.OutputType` to specify an input for another entry-point function. Pass the input by using the `codegen -args` option. Do not pass a `coder.OutputType` object as an input to a generated MEX function.

Creation

Syntax

```
t = coder.OutputType(func)
t = coder.OutputType(func,n)
```

Description

`t = coder.OutputType(func)` creates an object that is derived from the `coder.OutputType` class to represent the first output of the entry-point function `func`.

`t = coder.OutputType(func,n)` creates an object that is derived from the `coder.OutputType` class to represent the `n`-th output of the entry-point function `func`.

Input Arguments

`func` — Entry-point function name
character vector | string scalar

Name of entry-point function from which to define the output type.

Example: `coder.OutputType('myConstructor')`

n — Entry-point function output index
integer value

Index that indicates the n-th output variable of the corresponding entry-point function.

Example: `coder.OutputType('myFnWithTwoOutputs',1)`

Example: `coder.OutputType('myFnWithTwoOutputs',2)`

Properties

FunctionName — Entry-point function name
character vector | string scalar

Name of entry-point function from which the output type is derived.

OutputIndex — Entry-point function output index
integer value

Index of entry-point function output from which the output type is derived.

Examples

Use coder.OutputType to Represent a Variable-Size String Input

Suppose that you have a function `useString` that is intended to operate on a variable-size string input. Write a constructor function for a variable-size string. Pass the output as an input to `useString` by using `coder.OutputType`.

Write a MATLAB function `useString` that performs operations on an input string.

```
function y = useString(x)
%#codegen
y = replace(x,"be","not be");
end
```

To construct a variable-size input, write a constructor function.

```
function str = myConstructor(charArr)
%#codegen
str = string(charArr);
```

To generate code, specify an input type to the construction function. Declare a variable-size character vector input by using `coder.typeof`. Use `coder.OutputType` to represent the output type of the constructor function as the input type to the string operation function.

```
% get type of var-size char array bounded as 1-by-100
t = coder.typeof('a', [1 100], [0 1]);
% get output type
v = coder.OutputType('myConstructor');
% generate MEX function
codegen myConstructor -args {t} useString -args {v} -report -config:mex
```

Test the generated code by calling the MEX function in MATLAB:

```
a = myConstructor_mex('myConstructor','To be, or not to be.')
b = myConstructor_mex('useString',a)

a =
    "To be, or not to be."
b =
    "To not be, or not to not be."
```

Limitations

- You cannot use `coder.OutputType` in the field of a structure, cell, or in an array.

See Also

`codegen` | `coder.ArrayType` | `coder.ClassType` | `coder.Constant` | `coder.EnumType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` | `coder.typeof`

Topics

“Pass an Entry-Point Function Output as an Input”
“Generate Code for Multiple Entry-Point Functions”

Introduced in R2018b

coder.ReportInfo Properties

Code generation report information

Description

`coder.ReportInfo` properties contain information about code generation settings, input files, generated files, and code generation messages. All `coder.ReportInfo` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.ReportInfo` object. When you export code generation report information to a variable in your base MATLAB workspace, a `coder.ReportInfo` object is automatically created that contains this information. See "Access Code Generation Report Information Programmatically".

Properties

`Summary` — Summary of code generation
`coder.Summary` object

This property is read-only.

A summary of code generation including information about code generation success, path to the code generation output, toolchain, and build configuration, specified as a `coder.Summary` object. See coder.Summary Properties.

`Config` — Code generation configuration settings
`coder.MexCodeConfig` object | `coder.CodeConfig` object | `coder.EmbeddedCodeConfig` object

This property is read-only.

Code generation configuration settings, specified as a `coder.MexCodeConfig`, `coder.CodeConfig`, or `coder.EmbeddedCodeConfig` object.

`InputFiles` — Input files for code generation
array of `coder.CodeFile` objects | array of `coder.File` objects

This property is read-only.

Heterogeneous array containing descriptions of input files for code generation. Each element of the array contains a description of one input file.

- If an input file contains text, the corresponding array element is a `coder.CodeFile` object. See coder.CodeFile Properties.
- If an input file does not contain text (for example, a P-coded file), the corresponding array element is a `coder.File` object. See coder.File Properties.

`GeneratedFiles` — Generated files
array of `coder.CodeFile` objects | array of `coder.File` objects

This property is read-only.

Heterogeneous array containing descriptions of generated files. Each element of the array contains a description of one generated file.

- If the generated file contains text, the corresponding array element is a `coder.CodeFile` object. See coder.CodeFile Properties.
- If the generated file does not contain text (for example, a P-coded file), the corresponding array element is a `coder.File` object. See coder.File Properties.

`Functions` — MATLAB functions and methods used in code generation
array of `coder.Function` objects | array of `coder.Method` objects

This property is read-only.

Heterogeneous array containing descriptions of MATLAB functions and methods that are used in code generation. Each element of the array contains a description of one function or method.

- Array elements containing descriptions of functions are `coder.Function` objects. See coder.Function Properties.
- Array elements containing descriptions of methods are `coder.Method` objects. See coder.Method Properties.

`Messages` — Code generation error, warning, and informational messages
array of `coder.Message` objects

This property is read-only.

Array containing descriptions of error, warning, and informational messages produced during code generation. Each element of the array is a `coder.Message` object that contains a description of one message. See coder.Message Properties.

See Also

coder.CodeFile Properties | coder.File Properties | coder.Function Properties | coder.Message Properties | coder.Method Properties | coder.Summary Properties

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

coder.Summary Properties

Summary of code generation from MATLAB code

Description

`coder.Summary` properties contain a summary of code generation from MATLAB code. The summary includes information about code generation success, path to the code generation output, toolchain, and build configuration. All `coder.Summary` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.Summary` object. When you generate a `coder.ReportInfo` object that contains code generation report information, a `coder.Summary` object is automatically created as one of its properties. For more information, see coder.ReportInfo Properties and “Access Code Generation Report Information Programmatically”.

Properties

`Success` — Status of code generation

true | false

This property is read-only.

Status of code generation from MATLAB code, specified as one of the values in this table.

Value	Description
`true`	Code generation succeeded.
`false`	Code generation failed.

Data Types: `logical`

`Date` — Date and time of code generation

character vector

This property is read-only.

Date and time of code generation from MATLAB code, specified as a character vector.

Example: `'28-Nov-2018 09:59:54'`

Data Types: `char`

`OutputFile` — Path to output of code generation

character vector

This property is read-only.

Path to the generated files, specified as a character vector.

Example: `'C:\coder\R2019a\Report Info Object\codegen\lib\foo\foo.lib'`

Data Types: `char`

`Processor` — Information about processor used for code generation

character vector

This property is read-only.

Information about processor used for code generation from MATLAB code, specified as a character vector.

Example: `'Generic->MATLAB Host Computer'`

Data Types: `char`

`Version` — Version of MATLAB Coder used for code generation

character vector

This property is read-only.

Version of MATLAB Coder used for code generation from MATLAB code, specified as a character vector.

Example: `'MATLAB Coder 4.2 (R2019a)'`

Data Types: `char`

`Toolchain` — Toolchain used for code generation

character vector

This property is read-only.

Toolchain used for code generation from MATLAB code, specified as a character vector. This property is present only if you generate standalone code.

Example: `'Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)'`

Data Types: `char`

`BuildConfiguration` — Build configuration selected for code generation
character vector

This property is read-only.

Build configuration selected for code generation from MATLAB code, specified as a character vector. This property is present only if you generate standalone code.

Example: `'Faster Builds'`

Data Types: `char`

See Also

coder.ReportInfo Properties

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

coder.File Properties

Description of file without text that is involved in code generation

Description

`coder.File` properties contain the path and extension of a file that is involved in code generation from MATLAB code and does not have text (for example, a P-coded file). All `coder.File` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.File` object. A `coder.ReportInfo` object contains one `coder.File` object for every file involved in code generation that does not have text. These `coder.File` objects are automatically created when you export code generation report information to a `coder.ReportInfo` object. For more information, see coder.ReportInfo Properties and “Access Code Generation Report Information Programmatically”.

Descriptions of files that have text are stored in `coder.CodeFile` objects. See coder.CodeFile Properties.

Properties

`Path` — Path to the file

character vector

This property is read-only.

Path to a file that does not have text (for example, a P-coded file) and is involved in code generation from MATLAB code, specified as a character vector.

Example: `'C:\coder\R2019a\Report Info Object\foo.p'`

Data Types: `char`

`Extension` — Extension of the file

`'.p'` | ...

This property is read-only.

Extension of a file that does not have text (for example, a P-coded file) and is involved in code generation from MATLAB code, specified as a character vector.

Data Types: `char`

See Also

coder.CodeFile Properties | coder.ReportInfo Properties

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

coder.CodeFile Properties

Description of file containing text that is involved in code generation

Description

`coder.CodeFile` properties contain the text, path, and extension of a file that is involved in code generation from MATLAB code. All `coder.CodeFile` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.CodeFile` object. A `coder.ReportInfo` object contains one `coder.CodeFile` object for every file involved in code generation that has text. These `coder.CodeFile` objects are automatically created when you export code generation report information to a `coder.ReportInfo` object. For more information, see coder.ReportInfo Properties and “Access Code Generation Report Information Programmatically”.

Descriptions of files that do not have text are stored in `coder.File` objects. See coder.File Properties.

Properties

`Text` — Text of the file

character vector

This property is read-only.

Text of a file that is involved in code generation from MATLAB code, specified as a character vector.

Data Types: `char`

`Path` — Path to the file

character vector

This property is read-only.

Path to a file with text that is involved in code generation from MATLAB code, specified as a character vector.

Example: `'C:\coder\R2019a\Report Info Object\foo.m'`

Data Types: `char`

`Extension` — Extension of the file

`'.m'` | `'.mlx'` | `'.c'` | `'.cpp'` | `'.cu'` | `'.h'` | `'.hpp'` | `'.cuh'`

This property is read-only.

Extension of a file with text that is involved in code generation from MATLAB code, specified as a character vector.

Data Types: `char`

See Also

coder.File Properties | coder.ReportInfo Properties

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

coder.Function Properties

Description of MATLAB function used in code generation

Description

`coder.Function` properties contain the description of a MATLAB function that is used in code generation. All `coder.Function` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.Function` object. A `coder.ReportInfo` object contains one `coder.Function` object for every MATLAB function that is used in code generation. These `coder.Function` objects are automatically created when you export code generation report information to a `coder.ReportInfo` object. For more information, see coder.ReportInfo Properties and “Access Code Generation Report Information Programmatically”.

Properties

`Name` — Name of function
character vector

This property is read-only.

Name of a MATLAB function used in code generation, specified as a character vector.

Example: `'foo'`

Data Types: `char`

`Specialization` — Specialization identifier of a function
double

This property is read-only.

The specialization identifier of a MATLAB function used in code generation. A value of zero indicates that the function is not specialized. A positive value indicates that the function is specialized.

Data Types: `double`

`File` — MATLAB file containing the function
`coder.CodeFile` object | `coder.File` object

This property is read-only.

A description of the MATLAB file that contains the function used in code generation.

- If the file contains text, this property is a `coder.CodeFile` object. See coder.CodeFile Properties.
- If the file does not contain text (for example, a P-coded file), this property is a `coder.File` object. See coder.File Properties.

`StartIndex` — Index of the first character of the function in the text of the file
double

This property is read-only.

1-based index of the first character of the function in the text of the file. If the file does not contain text (for example, a P-coded file), this property is equal to `0`.

To manually inspect the code, the line and column numbers corresponding to `StartIndex` are useful. Use the `getLineColumn` function to obtain that information.

Data Types: `double`

`EndIndex` — Index of the last character of the function in the text of the file
double

This property is read-only.

1-based index of the last character of the function in the text of the file. If the file does not contain text (for example, a P-coded file), this property is equal to `0`.

To manually inspect the code, the line and column numbers corresponding to `EndIndex` are useful. Use the `getLineColumn` function to obtain that information.

Data Types: `double`

See Also

coder.CodeFile Properties | coder.File Properties | coder.ReportInfo Properties | `getLineColumn`

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

coder.Method Properties

Description of method in a MATLAB class used in code generation

Description

`coder.Method` properties contain the description of a method in a MATLAB class that is used in code generation. All `coder.Method` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.Method` object. A `coder.ReportInfo` object contains one `coder.Method` object for every method in the MATLAB classes that are used in code generation. These `coder.Method` objects are automatically created when you export code generation report information to a `coder.ReportInfo` object. For more information, see coder.ReportInfo Properties and "Access Code Generation Report Information Programmatically".

Properties

`Name` — Name of method
character vector

This property is read-only.

Name of a method in a MATLAB class used in code generation, specified as a character vector.

Example: `'foo'`

Data Types: `char`

`ClassName` — Name of class containing the method
character vector

This property is read-only.

Name of the class containing the method, specified as a character vector.

Example: `'MyClass'`

Data Types: `char`

`Specialization` — Specialization identifier of the method

double

This property is read-only.

The specialization identifier of the method. A value of zero indicates that the method is not specialized. A positive value indicates that the method is specialized.

Data Types: `double`

`ClassSpecialization` — Specialization identifier of the class containing the method

double

This property is read-only.

The specialization identifier of the class containing the method. A value of zero indicates that the class is not specialized. A positive value indicates that the class is specialized.

Data Types: `double`

`File` — MATLAB file containing the method

`coder.CodeFile` object | `coder.File` object

This property is read-only.

A description of the MATLAB file that contains the method used in code generation.

- If the file contains text, this property is a `coder.CodeFile` object. See coder.CodeFile Properties.
- If the file does not contain text (for example, a P-coded file), this property is a `coder.File` object. See coder.File Properties.

`StartIndex` — Index of the first character of the method in the text of the file

double

This property is read-only.

1-based index of the first character of the method in the text of the file. If the file does not contain text (for example, a P-coded file), this property is equal to `0`.

To manually inspect the code, the line and column numbers corresponding to `StartIndex` are useful. Use the `getLineColumn` function to obtain that information.

Data Types: `double`

`EndIndex` — Index of the last character of the method in the text of the file
double

This property is read-only.

1-based index of the last character of the method in the text of the file. If the file does not contain text, this property is equal to `0`.

To manually inspect the code, the line and column numbers corresponding to `EndIndex` are useful. Use the `getLineColumn` function to obtain that information.

Data Types: `double`

See Also

coder.CodeFile Properties | coder.File Properties | coder.ReportInfo Properties | `getLineColumn`

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

coder.Message Properties

Description of message produced during code generation

Description

`coder.Message` properties contain the description of an error, warning, or informational message that is produced during code generation from MATLAB code. All `coder.Message` properties are read-only. You can use dot notation to query these properties.

You do not directly create a `coder.Message` object. A `coder.ReportInfo` object contains one `coder.Message` object for every message produced during code generation. These `coder.Message` objects are automatically created when you export code generation report information to a `coder.ReportInfo` object. For more information, see coder.ReportInfo Properties and “Access Code Generation Report Information Programmatically”.

Properties

`Identifier` — Message identifier

character vector

This property is read-only.

The identifier associated with a message produced during code generation from MATLAB code, specified as a character vector.

Example: `'Coder:toolbox:unsupportedClass'`

Data Types: `char`

Type — Type of message

`'Error'` | `'Warn'` | `'Info'`

This property is read-only.

The type of a message produced during code generation from MATLAB code, specified as one of the values in this table.

Value	Description
`'Error'`	Error message
`'Warn'`	Warning message
`'Info'`	Informational message

Data Types: `char`

`Text` — Text of message

character vector

This property is read-only.

The text of a message produced during code generation from MATLAB code, specified as a character vector.

Example: `'Function 'svd' is not defined for values of class 'string'.'`

Data Types: `char`

`File` — MATLAB file containing code that caused the message

`coder.CodeFile` object | `coder.File` object

This property is read-only.

A description of the MATLAB file that contains code that caused the message.

- If the file contains text, this property is a `coder.CodeFile` object. See coder.CodeFile Properties.
- If the file does not contain text (for example, a P-coded file), this property is a `coder.File` object. See coder.File Properties.

`StartIndex` — Start index of code that caused the message

double

This property is read-only.

1-based index of the first character of the part of the file text that caused the message. If the file does not contain text (for example, a P-coded file), this property is equal to `0`.

To manually inspect the code, the line and column numbers corresponding to `StartIndex` are useful. Use the `getLineColumn` function to obtain that information.

Data Types: `double`

`EndIndex` — End index of code that caused the message
double

This property is read-only.

1-based index of the last character of the part of the file text that caused the message. If the file does not contain text (for example, a P-coded file), this property is equal to `0`.

To manually inspect the code, the line and column numbers corresponding to `EndIndex` are useful. Use the `getLineColumn` function to obtain that information.

Data Types: `double`

See Also

coder.CodeFile Properties | coder.File Properties | coder.ReportInfo Properties | `getLineColumn`

Topics

“Access Code Generation Report Information Programmatically”

Introduced in R2019a

Tools — Alphabetical List

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries and tables. You can use this tool to explore and choose a code replacement library or to view a predefined code replacement table. If you develop a custom code replacement library, you can use this viewer to verify table entries for the following properties:

- Argument order is correct.
- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct (highest priority is 0, and lowest priority is 100).
- Saturation or rounding mode specifications are not missing.

If you specify a library name when you open the viewer, the viewer displays the code replacement tables for that library. If you specify a table name when you open the viewer, the viewer displays the function and operator code replacement entries for that table. The viewer can only display code replacement tables that are defined. For more information on creating code replacement tables, see “Define Code Replacement Mappings” (Embedded Coder).

Abbreviated Entry Information

In the middle pane, the viewer displays entries that are in the selected code replacement table, along with abbreviated information for each entry.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`).

Field	Description
Implementation	Name of the implementation function, which can match or differ from **Name**.
NumIn	Number of input arguments.
In1Type	Data type of the first conceptual input argument.
In2Type	Data type of the second conceptual input argument.
OutType	Data type of the conceptual output argument.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
UsageCount	Not used.

Detailed Entry Information

In the middle pane, when you select an entry, the viewer displays entry details.

Field	Description
Description	Text description of the table entry (can be empty).
Key	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`), and the number of conceptual input arguments.
Implementation	Name of the implementation function, and the number of implementation input arguments.
Implementation type	Type of implementation: `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Saturation mode	Saturation mode that the implementation function supports. One of:
	`RTW_SATURATE_ON_OVERFLOW`
	`RTW_WRAP_ON_OVERFLOW`
	`RTW_SATURATE_UNSPECIFIED`

Field	Description

Rounding modes	Rounding modes that the implementation function supports. One or more of:
	`RTW_ROUND_FLOOR`
	`RTW_ROUND_CEILING`
	`RTW_ROUND_ZERO`
	`RTW_ROUND_NEAREST`
	`RTW_ROUND_NEAREST_ML`
	`RTW_ROUND_SIMPLEST`
	`RTW_ROUND_CONV`
	`RTW_ROUND_UNSPECIFIED`

GenCallback file	Not used.
Implementation header	Name of the header file that declares the implementation function.
Implementation source	Name of the implementation source file.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
Total Usage Count	Not used.
Entry class	Class from which the current table entry is instantiated.
Conceptual arguments	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), and data type for each conceptual argument.
Implementation	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), data type, and alignment requirement for each implementation argument.

Fixed-Point Entry Information

When you select an operator entry that specifies net slope fixed-point parameters, the viewer displays fixed-point information.

Field	Description
Net slope adjustment factor F	Slope adjustment factor (F) part of the net slope factor, $F2^E$, for net slope table entries. You use this factor with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Net fixed exponent E	Fixed exponent (E) part of the net slope factor, $F2^E$, for net slope table entries. You use this fixed exponent with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Slopes must be the same	Indicates whether code replacement request processing must check that the slopes on arguments (input and output) are equal. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.
Must have zero net bias	Indicates whether code replacement request processing must check that the net bias on arguments is zero. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.

Open the Code Replacement Viewer

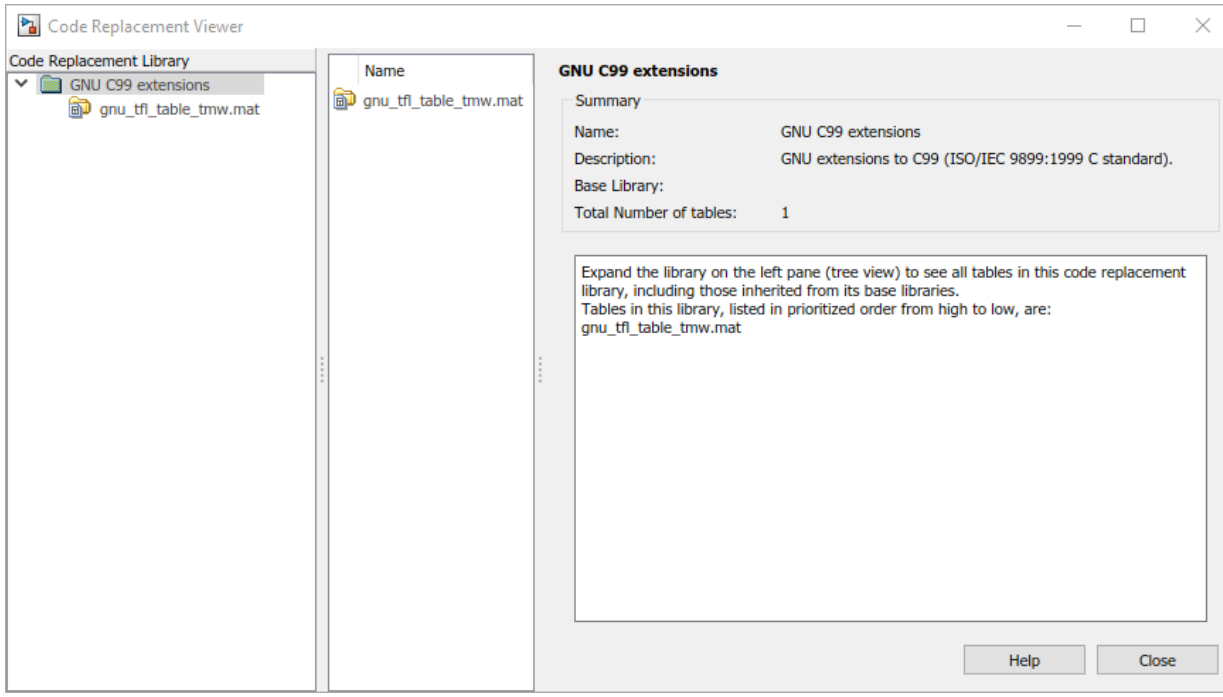
Open from the MATLAB command prompt using `crviewer`.

Examples

Display Contents of Code Replacement Library

This example opens the registered code replacement library `GNU C99 extensions`.

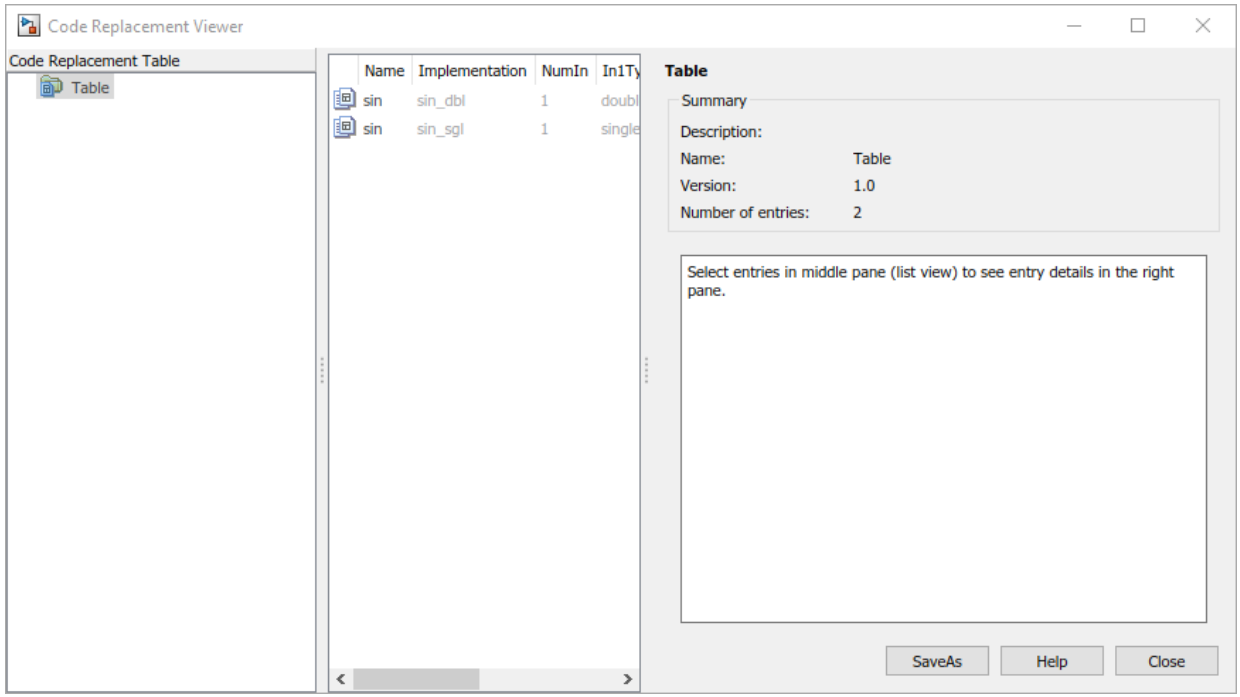
```
crviewer('GNU C99 extensions')
```



Display Contents of Code Replacement Table

This example opens a predefined code replacement table `crl_table_sinfcn`. To learn how to create this example table, see “Define Code Replacement Mappings” (Embedded Coder).

```
crviewer(crl_table_sinfcn)
```



- “Choose a Code Replacement Library”

Programmatic Use

`crviewer('library')` opens the Code Replacement Viewer and displays the contents of `library`, where `library` is a character vector that names a registered code replacement library.

`crviewer(table)` opens the Code Replacement Viewer and displays the contents of a predefined `table`, where `table` is a MATLAB file that defines code replacement tables. The table must be user predefined and the file must be in the current folder or on the MATLAB path.

See Also

Topics

“Choose a Code Replacement Library”
“What Is Code Replacement?”
“Code Replacement Libraries”
“Code Replacement Terminology”

Introduced in R2014b