# MATLAB® Coder™
Reference

MATLAB®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Trademarks

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

Patents

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

Revision History

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

Check Bug Reports for Issues and Fixes

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

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

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

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

- "C Code Generation Using the MATLAB Coder App"

Programmatic Use

`coder`

See Also

See Also

Apps
Fixed-Point Converter

Functions
`codegen`

Topics
"C Code Generation Using the MATLAB Coder App"

Function Reference

codegen

Generate C/C++ code from MATLAB code

Syntax

```
codegen options files fcn_1 args... fcn_n args
codegen project_name
```

Description

`codegen options files fcn_1 args... fcn_n args` translates the MATLAB functions *fcn_1* through *fcn_n* to a C/C++ static or dynamic library, executable, or MEX function. Optionally, you can specify custom *files* to include in the build. `codegen` applies the *options* to functions *fcn_1* through *fcn_n*. It applies *args* to the preceding function only (*fcn_n*). If you specify C++, MATLAB Coder™ wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

`codegen project_name` generates output for the MATLAB Coder project `project_name`. `codegen` generates a MEX function, C/C++ static or dynamic library, or C/C++ executable depending on the project settings that you defined for `project_name`.

By default, `codegen` generates files in the folder codegen/*target*/*fcn_name*.

target can be:

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

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

`codegen` copies the MEX function and executable file to the current working folder or to the output folder that the `-d` option specifies.

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 them to a different location

Input Arguments

args

args applies only to the preceding function, *fcn_n*.

`-args` *example_inputs*	Define the size, class, and complexity of MATLAB function inputs by providing a cell array of example input values. The position of the example input in the cell array must correspond to the position of the input argument in the MATLAB function definition. 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.
	Specify the example inputs immediately after the function to which they apply.
	Instead of an example value, you can provide a `coder.Type` object. To create a `coder.Type` object, use `coder.typeof`.

fcn_1

`fcn_1...` `fcn_n` are the MATLAB entry-point functions from which to generate a MEX function, C/C++ library, or C/C++ executable code. In most cases, you have only one function. Make sure that *fcn_1... fcn_n* are suitable for code generation.

If these MATLAB functions are in files on a path that contains non 7-bit ASCII characters, such as Japanese characters, it is possible that `codegen` does not find them.

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

files

Space-separated list of custom files to include in generated code. You can include the following 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`)

If these files are on a path that contains non 7-bit ASCII characters, such as Japanese characters, it is possible that `codegen` does not find them.

options

Choice of compiler options. `codegen` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the rightmost option prevails.

`-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 a Fixed-Point Designer license.
`-config` *config_object*	Specify the configuration object that contains the code generation parameters. *config_object* is based on one of the following classes:

- `coder.CodeConfig` — Parameters for standalone C/C++ library or executable generation if no Embedded Coder license is 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 an Embedded Coder license 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')
```

-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.

You must have a Fixed-Point Designer license 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'; specifies that myadd_test is the test file for the floating-point to fixed-point configuration object fixptcfg.

You must have a Fixed-Point Designer license 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.

-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.
	Spaces in *include_path* can lead to code generation failures in certain operating system configurations. 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.
-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.
-nargout *number_args*	Specify the number of output arguments in the generated entry-point function. The code generator produces the specified number of output arguments in the order in which they occur in the MATLAB function definition.

-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.
	Specify `-O` at the command line once for each optimization.
	If not specified, `codegen` uses inlining and OpenMP for optimization.
`-report`	Generate 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 `-launchreport` option.
	If you have an Embedded Coder license, this option also enables the Code Replacements and Static Code Metrics reports.
`-singleC`	Generate single-precision C/C++ code.
	You must have a Fixed-Point Designer license 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'`. 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.

project_name

Name of the MATLAB Coder project that you want `codegen` to build. The project name must not contain spaces.

Examples

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

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 indicates that the function
% is intended for code generation
r = rand();
```

2   Generate and run the MEX function. By default, `codegen` names the generated MEX function coderand_mex.

```
codegen coderand
coderand_mex
```

Generate C executable files from a MATLAB function that is suitable for code generation. Specify the main C function as a configuration parameter.

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

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

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

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_initialize.h"
#include "coderand_terminate.h"
int main()
{
    coderand_initialize();

    printf("coderand=%g\n", coderand());

    coderand_terminate();

    return 0;
}
```

3 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 coderand
```

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

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.

Generate C library files in a custom folder from a MATLAB function with inputs of different classes and sizes. The first input is a 1-by-4 vector of unsigned 16-bit integers. The second input is a double-precision scalar.

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

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

2   Generate the C library files in a custom folder `mcaddlib` using the `-config:lib` option.

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

Generate C library files from a MATLAB function that takes a fixed-point input.

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

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

2   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`.

Specify global data at the command line.

1   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;
```

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];
```
Compile the function to generate a MEX file named `use_globalsx`.

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

Generate output for a MATLAB Coder project, `test_foo.prj`, that includes one file, `foo.m`, and has it output type set to C/C++ Static Library.

```
codegen test_foo.prj
```

`codegen` generates a C library, `foo`, in the `codegen\lib\foo` folder.

Generate a MEX function for a function, `displayState`, that has an input parameter that is an enumerated type.

1 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
```

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

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

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

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

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

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

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

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

4   Generate a MEX function for `displayState`.

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

Convert floating-point MATLAB code to fixed-point C code

This example requires a Fixed-Point Designer license.

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

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

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

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

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

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

4 Set the test bench name.

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

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

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

6 Generate the code using the `-float2fixed` option.

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

Convert double-precision MATLAB code to single-precision C code.

This example requires a Fixed-Point Designer license.

Suppose that `myfunction` takes two double scalar inputs. Use the `-singleC` option to generate single-precision C/C++ code.

```matlab
codegen -singleC myfunction -args {1 2}
```

Tutorials

- "C Code Generation at the Command Line"
- "C Code Generation Using the MATLAB Coder App"
- "MEX Function Generation at the Command Line"

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

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

Topics

"C Code Generation at the Command Line"
"C Code Generation Using the MATLAB Coder App"
"MEX Function Generation at the Command Line"
"Code Generation"
"Specify Properties of Entry-Point Function Inputs"
"Specify Cell Array Inputs at the Command Line"
"Specify Global Cell Arrays at the Command Line"
"Specify a Language for Code Generation"
"Paths and File Infrastructure Setup"
"Generate Code for Global Data"
"Synchronizing Global Data with MATLAB"
"Generate Code for Multiple Entry-Point Functions"
"Convert MATLAB Code to Fixed-Point C Code"
"Generate Single-Precision C Code at the Command Line"
"Generate Single-Precision MATLAB Code"

External Websites

"Control Compilation of parfor-Loops"

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
```

- "C Code Generation 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"

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

See Also
`codegen` | MATLAB Coder

Topics
"C Code Generation 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
`pcode` | `codegen`

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]);
```

- "Replace the exp Function with a Lookup Table"
- "Replace a Custom Function with a Lookup Table"

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 single 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

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

Package: coder

Call external C/C++ function

Syntax

```
coder.ceval('cfun_name')
coder.ceval('cfun_name', cfun_arguments)
cfun_return = coder.ceval('cfun_name')
cfun_return = coder.ceval('cfun_name', cfun_arguments)
coder.ceval('-global','cfun_name',cfun_arguments)
cfun_return=coder.ceval('-global','cfun_name',cfun_arguments)
```

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.

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.

cfun_return = coder.ceval('*cfun_name*') 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.

cfun_return = coder.ceval('*cfun_name*', *cfun_arguments*) executes *cfun_name* with arguments *cfun_arguments* and returns *cfun_return*.

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

cfun_return=coder.ceval('-global','*cfun_name*',*cfun_arguments*)

For code generation, you must specify the type, size, and complexity data type of return values and output arguments before calling coder.ceval.

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. In this case, if you do not use the `coder.ref`, `coder.rref`, and `coder.wref` constructs, a copy of the argument might appear in the generated code to enforce MATLAB semantics for arrays.

If you pass a global variable by reference using `coder.ref`, `coder.rref` or `coder.wref`, and the custom C code saves the address of this global variable, use the `-global` flag to synchronize for the variables passed to the custom C code. Synchronization occurs before and after calls to the custom code. If you do not synchronize global variables under these circumstances and the custom C code saves the address and accesses it again later, the value of the variable might be out of date.

Note: The `-global` flag does not apply for MATLAB Function blocks.

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

Use `coder.ceval` only in MATLAB for code generation. `coder.ceval` generates an error in uncompiled MATLAB code. Use `coder.target` to determine if the MATLAB function is executing in MATLAB. If it is, do not use `coder.ceval` to call the C/C++ function. Instead, call the MATLAB version of the C/C++ function.

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 in a `coder.ceval` call. 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.

Examples

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

1. 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`.

```
#ifdef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

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

2. Write the C function `foo.c`.

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

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

3. Write a function `callfoo` that calls `foo` using `coder.ceval`.

```
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
    y = coder.ceval('foo', 10, 20);
end
end
```

4. Generate C library code for function `callfoo`, passing `foo.c` and `foo.h` as parameters to include this custom C function in the generated code.

```
codegen -config:lib callfoo foo.c foo.h
```
`codegen` generates C code in the `codegen\lib\callfoo` subfolder.

```
double callfoo(void)
{
  /*  Executing in generated code, call C function foo */
  return foo(10.0, 20.0);
```

```
}
```

In this case, you have not specified the type of the input arguments, that is, the type of the constants `10` and `20`. Therefore, the arguments are implicitly of double-precision, floating-point type by default, because the default type for constants in MATLAB is `double`.

Call a C library function from MATLAB code:

1 Write a MATLAB function `absval`.

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

2 Generate the C library for `absval.m`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib absval -args {0.0}
```
`codegen` creates the library `absval.lib` and header file `absval.h` in the folder /`codegen/lib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the same folder.

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

```
function y = callabsval   %#codegen
y = -2.75;
% Check the target. Do not use coder.ceval if callabsval is
% executing in MATLAB
if coder.target('MATLAB')
  % Executing in MATLAB, call function absval
  y = absval(y);
else
  % Executing in the generated code.
  % Call the initialize function before calling the
  % C function for the first time
  coder.ceval('absval_initialize');

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

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

4 Convert the code in `callabsval.m` to a MEX function so you can call the C library function `absval` directly from MATLAB.

```
codegen -config:mex callabsval codegen/lib/absval/absval.lib...
    codegen/lib/absval/absval.h
```

5 Call the C library by running the MEX function from MATLAB.

```
callabsval_mex
```

Tutorials

• "Return Multiple Values from C Functions"

See Also

```
coder.ref | coder.rref | coder.wref | coder.target | codegen | | | |
coder.extrinsic | coder.opaque | | |
```

Topics

"Return Multiple Values from C Functions"
"Compilation Directive %#codegen"
"External Code Integration"
"Data Definition Basics"

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);
end
```

Open the Configuration Parameters dialog box.

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

Specify the locations of `myMult2.h` and `myMult2.c` for simulation and code generation.

- On the **Simulation Target** pane, under **Additional build information**, for **Include directories**, enter `mycfiles`.
- For **Source files**, enter `mycfiles\myMult2.c`.
- On the **Code Generation > Custom Code** pane, select the **Use the same custom code settings as Simulation Target** check box.

Save the model as `mymodel`.

Build the model.

The file `mymodel.h` contains this statement:

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

Input Arguments

`headerfile` — Name of header file
character vector

Name of a header file specified as a character vector. `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

See Also
codegen | coder.ceval | coder.config | coder.target

Topics
"Specify External File Locations"

Introduced in R2013a

coder.config

Package: coder

Create MATLAB Coder code generation configuration objects

Syntax

```
config_obj = coder.config
config_obj = coder.config('mex')
config_obj = coder.config('lib')
config_obj = coder.config('dll')
config_obj = coder.config('exe')
config_obj = coder.config(c_output_type,'ecoder',false)
config_obj = coder.config(c_output_type,'ecoder',true)
config_obj = coder.config('fixpt')
config_obj = coder.config('single')
```

Description

`config_obj = coder.config` creates a `coder.MexCodeConfig` code generation configuration object for use with `codegen` when generating a MEX function.

`config_obj = coder.config('mex')` creates a `coder.MexCodeConfig` code generation configuration object for use with `codegen` when generating a MEX function.

`config_obj = coder.config('lib')` creates a code generation configuration object for use with `codegen` when generating a C/C++ static library. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` configuration object.

`config_obj = coder.config('dll')` creates a code generation configuration object for use with `codegen` when generating a C/C++ dynamic library. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` configuration object.

`config_obj = coder.config('exe')` creates a code generation configuration object for use with `codegen` when generating a C/C++ executable. If the Embedded Coder

product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` configuration object.

`config_obj = coder.config(`*`c_output_type`*`,'ecoder',false)` creates a `coder.CodeConfig` configuration object to generate *`c_output_type`* even if the Embedded Coder product is installed. *`c_output_type`* is `'lib'`, `'dll'`, or `'exe'`.

`config_obj = coder.config(`*`c_output_type`*`,'ecoder',true)` creates a `coder.EmbeddedCodeConfig` configuration object to generate *`c_output_type`* even if the Embedded Coder product is not installed. However, code generation using a `coder.EmbeddedCodeConfig` object requires an Embedded Coder license. *`c_output_type`* is `'lib'`, `'dll'`, or `'exe'`.

`config_obj = coder.config('fixpt')` creates a `coder.FixptConfig` configuration object for use with `codegen` when generating fixed-point C/C++ code from floating-point MATLAB code. Creation of a `coder.FixptConfig` code configuration object requires the Fixed-Point Designer product.

`config_obj = coder.config('single')` creates a `coder.SingleConfig` configuration object for use with `codegen` when generating single-precision MATLAB code from double-precision MATLAB code. Creation of a `coder.SingleConfig` code configuration object requires the Fixed-Point Designer product.

Examples

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   Enable the code generation report.

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

4 Generate a MEX function in the current folder specifying the configuration object using the -config option.

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

Create a code generation configuration object to generate 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 even if the Embedded Coder product is installed .

```
cfg = coder.config('lib','ecoder',false)
% Returns a coder.CodeConfig object even if the Embedded
% Coder product is installed.
```

Create a floating-point to fixed-point conversion configuration object.

```
fixptcfg = coder.config('fixpt');
% Returns a coder.FixptConfig object
```

Create a double-precision to single-precision conversion configuration object.

```
scfg = coder.config('single');
% Returns a coder.SingleConfig 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

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

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 statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((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 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 O
```

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 O
```
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.

```matlab
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.

```matlab
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

- The code generator constant-folds expressions automatically when possible. 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.

See Also

Topics
"Constant Folding"

Introduced in R2013b

coder.cstructname

Package: coder

Name structure in generated code

Syntax

```
coder.cstructname(var,'structName')
coder.cstructname(var,'structName','extern')
coder.cstructname(var,'structName','extern',Name,Value)
newt = coder.cstructname(t,'structName')
newt = coder.cstructname(t,'structName','extern')
newt = coder.cstructname(t,'structName','extern',Name,Value)
```

Description

coder.cstructname(var,'structName') specifies the name of the structure
type that represents var in the generated C/C++ code. var is a structure or cell array
variable. structName is the name for the structure type in the generated code. Use this
syntax in a function from which you generate code. Call coder.cstructname before the
first use of the variable. If var is a cell array element, call coder.cstructname after
the first assignment to the element.

coder.cstructname(var,'structName','extern') declares an externally defined
structure. It does not generate the definition of the structure type. Provide the definition
in a custom include file.

coder.cstructname(var,'structName','extern',Name,Value) uses additional
options specified by one or more Name,Value pair arguments.

newt = coder.cstructname(t,'structName') returns a copy of the type object
specified by t. The copy specifies the name structName to use for the structure
type that represents t in the generated code. t is a coder.StructType object or a
coder.CellType object. Use this syntax to create a type that you use with the codegen
-args option. You cannot use this syntax in a function from which you generate code.

newt = coder.cstructname(t,'structName','extern') returns a coder.type that uses an externally defined structure. Provide the structure definition in a custom include file.

newt = coder.cstructname(t,'structName','extern',Name,Value) uses additional options specified by one or more Name,Value pair arguments.

Limitations

- You cannot use coder.cstructname with global variables.

- If var is a cell array or t is a coder.CellType object, the field names of externally defined structures must be f1, f2, and so on.

- If var is a cell array element, call coder.cstructname after the first assignment to the element. For example:

```
...
x = cell(2,2);
x{1} = struct('a', 3);
coder.cstructname(x{1}, 'mytype');
...
```

Tips

- The code generator represents a heterogeneous cell array as a structure in the generated C/C++ code. To specify the name of the generated structure type, use coder.cstructname.

- Using coder.cstructname with a homogeneous coder.CellType object t makes the returned object heterogeneous unless t is permanently homogeneous. If the makeHomogeneous method created t or if t is variable size, t is permanently homogeneous.

- When used with a coder.CellType object, coder.cstructname creates a coder.CellType object that is permanently heterogeneous.

- In a function from which you generate code, using coder.cstructname with a cell array variable makes the cell array heterogeneous. Unless the cell array type is permanently set to homogeneous, you can use coder.cstructname with an entry-point function input that is a cell array.

- To use `coder.cstructname` on arrays, use single indexing. For example, you cannot use `coder.cstructname(x(1,2))`. Instead, use single indexing, for example `coder.cstructname(x(n))`.

- If you use `coder.cstructname` on an array, it sets the name of the base type of the array, not the name of the array. Therefore, you cannot use `coder.cstructname` on the base element and then on the array. For example, the following code does not work. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previous `coder.cstructname`, `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');
```

- If you are using custom structure types, specify the name of the header file that includes the external definition of the structure. Use the `HeaderFile` input argument.

- 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. To take advantage of target-specific function implementations that require data to be aligned, use the `Alignment` input argument.

- You can also use `coder.cstructname` to assign a name to a substructure, which is a structure that appears as a field of another structure. For more information, see "Assign a Name to a SubStructure" on page 2-62.

Input Arguments

structName

The name of the structure type in the generated code.

t

`coder.StructType` object or `coder.CellType` object.

var

Structure or cell array variable.

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'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. Hence it is not matched by CRL functions that require alignment.

`Alignment` must be either `-1` or a power of `2` that is not greater than `128`.

Default: -1

`'HeaderFile'`

Name of the header file that contains the external definition of the structure, for example, `'mystruct.h'`. Specify the path to the file. Use 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.

Output Arguments

newt

`coder.StrucType` object or `coder.CellType` object.

Examples

Apply `coder.cstructname` to Top-Level Inputs

Generate code for a MATLAB function that takes structure inputs.

1   Write a MATLAB function, `topfun`, that assigns the name `MyStruct` to its input parameter.

```
function y = topfun(x)    %#codegen
% Assign the name 'MyStruct' to the input variable in
% the generated code
  coder.cstructname(x, 'MyStruct');
  y = x;
end
```

2   Declare a structure `s` in MATLAB. `s` is the structure definition for the input variable `x`.

```
s = struct('a',42,'b',4711);
```

3   Generate a MEX function for `topfun`, using the `-args` option to specify that the input parameter is a structure.

```
codegen topfun.m -args { s }
```

`codegen` generates a MEX function in the default folder `codegen\mex\topfun`. In this folder, the structure definition is in `topfun_types.h`.

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

Assign a Name to a Structure and Pass It to a Function

Assign the name `MyStruct` to the structure `var`. Pass the structure to a C function `use_struct`.

1   Create a C header file, `use_struct.h`, for a `use_struct`function that takes a parameter of type `MyStruct`. Define a structure of type `MyStruct` in the header file.

```
#ifdef MATLAB_MEX_FILE
#include <tmwtypes.h>
```

```
#else
#include "rtwtypes.h"
#endif

typedef struct MyStruct
{
    double s1;
    double s2;
} MyStruct;

void use_struct(struct MyStruct *my_struct);
```

2 Write the C function use_struct.c.

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

#include "use_struct.h"

void use_struct(struct MyStruct *my_struct)
{
  double x = my_struct->s1;
  double y = my_struct->s2;
}
```

3 Write a m_use_struct compliant with MATLAB that declares a structure. Have the function assign the name MyStruct to the structure. Then, have the function call the C function use_struct using coder.ceval.

```
function m_use_struct    %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Declare a MATLAB structure
var.s1 = 1;
var.s2 = 2;

% Assign the name MyStruct to the structure variable.
% extern indicates this is an externally defined
% structure.
coder.cstructname(var, 'MyStruct', 'extern');

% Call the C function use_struct. The type of var
% matches the signature of use_struct.
% Use coder.rref to pass the variable var by
% reference as a read-only input to the external C
```

```
% function use_struct
coder.ceval('use_struct', coder.rref(var));
```

4 Generate C library code for function m_use_struct, passing use_struct.h to include the structure definition.

```
codegen -config:lib m_use_struct use_struct.c use_struct.h
```
codegen generates C code in the default folder codegen\lib\m_use_struct. The generated header file m_use_struct_types.h in this folder does not contain a definition of the structure MyStruct because MyStruct is an external type.

Assign a Name to a SubStructure

Use coder.cstructname to assign a name to a substructure.

1 Define a MATLAB structure, top, that has another structure, lower, as a field.

```
% Define structure top with field lower,
% which is a structure with fields a and b
top.lower = struct('a',1,'b',1);
top.c = 1;
```

2 Define a function, MyFunc, which takes an argument, TopVar, as input. Mark the function for code generation using %#codegen.

```
function out = MyFunc(TopVar) %#codegen
```

3 Inside MyFunc, include the following lines

```
coder.cstructname(TopVar,'topType');
coder.cstructname(TopVar.lower,'lowerType');
```

4 So that TopVar has the same type as top, generate C code for MyFunc with an argument having the same type as top.

```
codegen -config:lib MyFunc -args coder.typeof(top)
```

In the generated C code, the field variable TopVar.lower is assigned the type name lowerType. For instance, the structure declaration of the variable TopVar.lower appears in the C code as:

```
typedef struct
{
    /* Definitions of a and b appear here */
} lowerType;
```

and the structure declaration of the variable TopVar appears as:

```
    typedef struct
    {
         lowerType lower;
       /* Definition of c appears here */
    } topType;
```

Create a coder.type Object

Create a `coder.type` object and pass it as `codegen` argument.

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

In this example, you create a `coder.type` object T. The object is passed as a `codegen` argument. However, because of the `coder.cstructname` statement, T is replaced with `mytype` in the generated C code. For instance, the declaration of T appears in the C code as:

```
typedef struct
{
    /* Field definitions appear here */
} mytype;
```

Create a coder.type Object Using an Externally Defined Type

Create a C header file, `MyFile.h`, containing the definition of a structure type, `mytype`.

```
typedef struct {
    /* Field definitions */
    double a;
    float b;
    } mytype;
```

Save the file in the folder, `C:\MyHeaders`.

Define a `coder.type` object, T, with the same fields as `mytype`.

```
T = coder.typeof(struct('a',double(0),'b',single(0)));
```

Using `coder.cstructname`, rename T as `mytype`. Specify that the definition of `mytype` is in `MyFile.h`.

```
T = coder.cstructname(T,'mytype','extern','HeaderFile','MyFile.h');
```

Generate code for MATLAB function, `MyFunc`, which takes a structure of type, `T`, as input argument. Add the folder, `C:\MyHeaders`, to the include path during code generation.

```
codegen -config:lib MyFunc -args T -I C:\MyHeaders
```

In the generated code, the structure, `T`, is assigned the name, `mytype`. The code generator does not generate the definition of `mytype`. Instead the software includes the header file, `MyFile.h`, in the generated code.

Assign a Structure Type Name to a `coder.CellType` Object

Create a `coder.CellType` object for a cell array whose first element is char and whose second element is double.

```
T = coder.typeof({'c', 1})

T =

coder.CellType
   1x2 heterogeneous cell
       f0: 1x1 char
       f1: 1x1 double
```

`T` is a heterogeneous `coder.CellType` object.

Create a copy of `T` that specifies the name `myname` for the structure type that represents `T` in the generated code.

```
T = coder.cstructname(T, 'myname')

coder.CellType
   1x2 heterogeneous cell myname
       f0: 1x1 char
       f1: 1x1 double
```

Assign a Name to a Structure That Is an Element of a Cell Array

Write a function `struct_in_cell` that has a cell array `x{1}` that contains a structure. The `coder.cstructname` call follows the assignment to `x{1}`.

```
function z = struct_in_cell()
```

```
x = cell(2,2);
x{1} = struct('a', 3);
coder.cstructname(x{1}, 'mytype');
z = x{1};
end
```

Generate a static library for struct_in_cell.

```
codegen -config:lib struct_in_cell -report
```

The type for a has the name mytype.

```
 typedef struct {
  double a;
} mytype;
```

See Also

See Also
coder.StructType | coder.CellType | codegen | coder.ceval | coder.rref

Topics
"Code Generation for Cell Arrays"

Introduced in R2011a

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.

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.screener | coder.ceval

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

See Also
codegen | coder.resize | coder.runTest | coder.typeof

Topics
"Specify Properties of Entry-Point Function Inputs"

Introduced in R2012a

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.
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.

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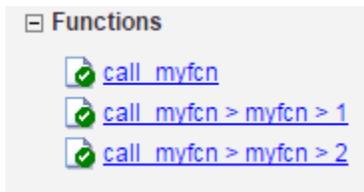
The code generator creates `call_myfcn>myfcn>1` for `mode` with a value of `'mode1'`. It creates `call_myfcn>myfcn>2` for `mode` with a value of `'mode2'`.

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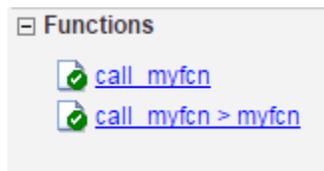
In the generated C code, you see one function for `my_fcn`.

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

See Also
`coder.inline`

Topics
"Force Code Generator to Use Run-Time Recursion"
"Compile-Time Recursion Limit Reached"

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 of the current function in generated code.

`coder.inline('never')` prevents inlining of the current function in generated code. For example, you may want to prevent inlining to simplify the mapping between the MATLAB source code and the generated code.

`coder.inline('default')` uses internal heuristics to determine whether or not to inline the current function.

In most cases, the heuristics used produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

Place the `coder.inline` directive inside the function to which it applies. The code generator does not inline entry-point functions.

`coder.inline('always')` does not inline functions called from `parfor`-loops. The code generator does not inline functions into `parfor`-loops.

Examples

- "Preventing Function Inlining" on page 2-80
- "Using coder.inline In Control Flow Statements" on page 2-80

Preventing Function Inlining

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

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

Using 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 you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
   coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
   coder.inline('never');
end

if any(divisor == 0)
   error('Can not divide by 0');
end

y = dividend / divisor;
```

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.

Introduced in R2011a

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

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'`

Data Types: `char`

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

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

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

Data Types: `char`

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

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

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

Data Types: `char`

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

- `coder.load` does not support loading objects.
- 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

See Also
`matfile` | `regexp` | `save`

Topics
"Regular Expressions" (MATLAB)

Introduced in R2013a

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)
```

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`.

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`. Enclose the class name in single quotes. `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 single 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
```

Alternatives

`coder.typeof`

See Also

coder.Type | coder.ArrayType | coder.EnumType | coder.FiType | coder.PrimitiveType | coder.StructType | coder.CellType | `codegen` | `coder.resize` | coder.ClassType

Topics

Introduced in R2011a

coder.nullcopy

Package: coder

Declare uninitialized variables

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. Preallocates memory for X without incurring the overhead of initializing memory.

`coder.nullcopy` does not support MATLAB classes as inputs.

Use With Caution

Use this function with caution. See "How to Eliminate Redundant Copies by Defining Uninitialized Variables".

Examples

The following example shows how to declare variable X as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo

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.

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`.

```matlab
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

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 *'

Data Types: char

value — Initial value of variable
character vector

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'`

Data Types: `char`

`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

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'`

Data Types: `char`

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 2-99

- 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

See Also
`coder.ceval | coder.ref | coder.rref | coder.wref`

Topics
"Specify Build Configuration Parameters"

Introduced in R2011a

coder.ref

Pass argument by reference

Syntax

```
coder.ref(arg)
```

Description

`coder.ref(arg)` passes `arg` by reference to an external C/C++ function that you call by using `coder.ceval`. 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.

Examples

Pass Scalar Variable by Reference

Pass a variable by reference to a C function that adds 1 to the variable.

Suppose that you have a C function `addone`.

```
void addone(double* p) {
  *p = *p + 1;
}
```

The C function prototype defines the input as a pointer because it is passed by reference.

Pass a variable by reference to `addone`.

```
...
u = 42;
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`.

```
void incr_struct(struct MyStruct *my_struct)
{
  my_struct->f1 =  my_struct->f1 + 1;
  my_struct->f2 =  my_struct->f2 + 1;
}
```

Suppose that a C header file, `MyStruct.h`, defines a structure type named `MyStruct`.

```
...
typedef struct MyStruct
{
    double f1;
    double f2;

} MyStruct;

void incr_struct(struct MyStruct *my_struct);

...
```

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 a C header file
`MyStruct.h`, use `coder.cstructname`.

```
...
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'MyStruct', 'extern', 'HeaderFile', 'MyStruct.h');
coder.ceval('incr_struct', coder.ref(s));
...
```

Pass Structure Field by Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.ref(s.s1.a));
...
```

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 setter or getter method, you cannot pass the property by reference because you cannot access the address of the property. Only the getter or setter method can directly read from or write to this property.

Tips

*   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 "External Function Calls from Generated Code".

See Also

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.rref` | `coder.wref`

Topics

"External Function Calls from Generated Code"
"Call External Functions with coder.ceval"

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 single 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
```

See Also

codegen | coder.typeof | coder.newtype

Topics

Introduced in R2011a

coder.rref

Pass argument by reference as read-only input

Syntax

```
coder.rref(arg)
```

Description

`coder.rref(arg)` passes `arg` by reference to an external C/C++ function that you call by using `coder.ceval`. Use `coder.rref` inside a `coder.ceval` call only. The code generator assumes that the variable that you pass in `coder.rref` is read-only. To avoid unpredictable results, the C/C++ function must not write to this variable.

Use a separate `coder.rref` construct for each argument that you pass by reference to the function.

Examples

Pass Scalar Variable as a Read-Only Reference

Pass a variable as a read-only reference to a C function that returns the value of the variable plus one.

Consider the C function `addone`:

```
double addone(const double* p) {
  return *p + 1;
}
```

The C function prototype defines the input as a pointer because it is passed by reference.

Pass a variable by reference to `addone`.

...

```
u = 42;
y = 0;
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`.

```
double use_struct(struct MyStruct *my_struct)
{
  return  my_struct->f1 + my_struct->f2;

}
```

Suppose that a C header file, `MyStruct.h`, defines a structure type named `MyStruct`.

```
...
typedef struct MyStruct
{
    double f1;
    double f2;
```

```
} MyStruct;

double use_struct(struct MyStruct *my_struct);

...
```

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 a C header file `MyStruct.h`, use `coder.cstructname`.

```
...
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'MyStruct', 'extern', 'HeaderFile', 'MyStruct.h');
y = 0;
y = coder.ceval('use_struct', coder.rref(s));
...
```

Pass Structure Field as a Read-Only Reference

```
...
s = struct('s1', struct('a', [0 1]));
y = coder.ceval('foo', coder.rref(s.s1.a));
...
```

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 setter or getter method, you cannot pass the property by reference because you cannot access the address of the property. Only the getter or setter method can directly read from or write to this property.

Tips

- 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 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 "External Function Calls from Generated Code".

See Also

See Also
`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.wref`

Topics
"External Function Calls from Generated Code"
"Call External Functions with coder.ceval"

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 =
```

2-121

```
  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

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'`

Data Types: `char`

`fcn` — Name of MATLAB function to replace
character vector

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'`

Data Types: `char`

`fcns` — Names of MATLAB functions to replace
character vector | 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.

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'}`

Data Types: `char` | `cell`

`mexfcn` — MEX function name
character vector

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'`

Data Types: `char`

`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

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.

fcn_1,...,fcn_n

Comma-separated list of names of entry-point MATLAB functions that you want to analyze. Specify as character vectors.

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.

Analyze the MATLAB function `foo1` that calls `foo2`. Put `foo1` and `foo2` in separate files.

```
function out = foo1(in)
  out = foo2(in);
  disp(out);
end

function out = foo2(in)
  out = eval(in);
end

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.
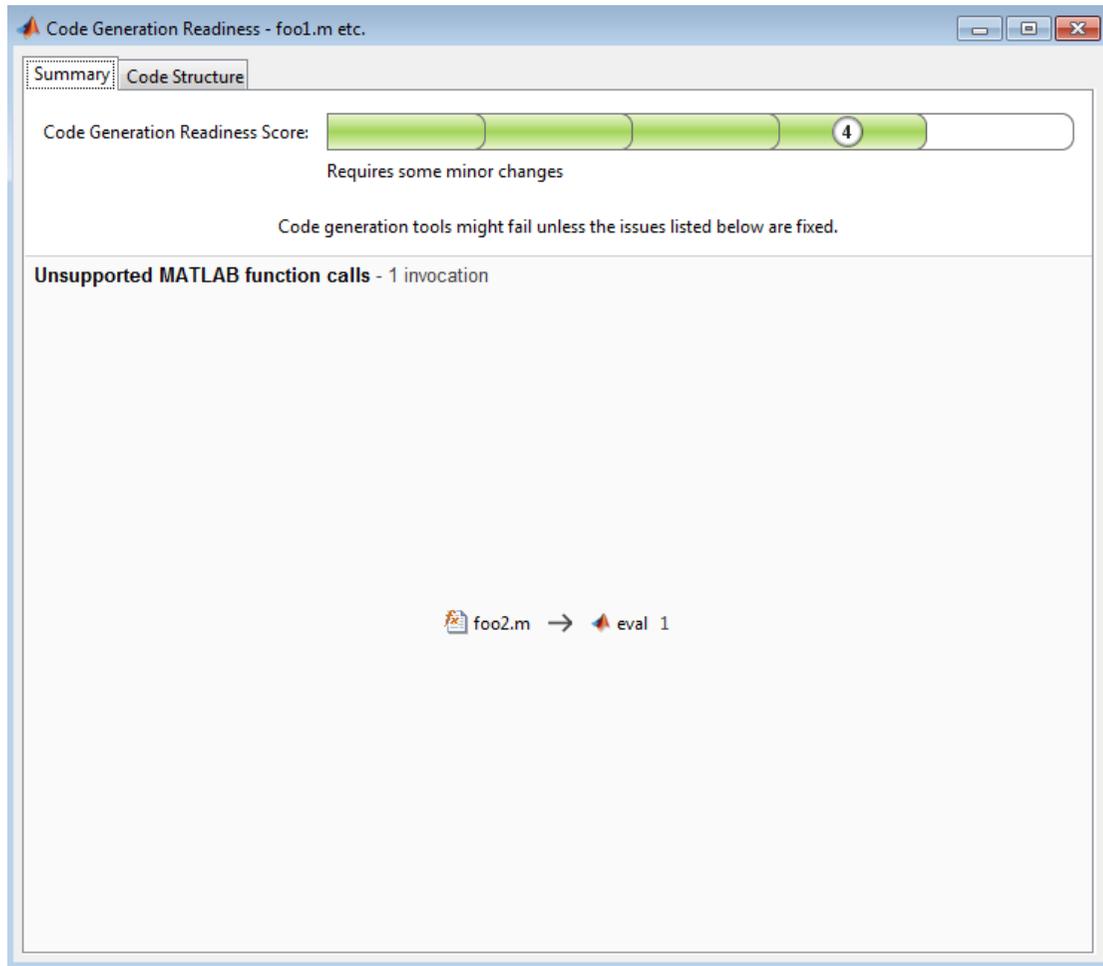
In the report, click the **Code Structure** tab and select the **Show MATLAB functions** check box.

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.
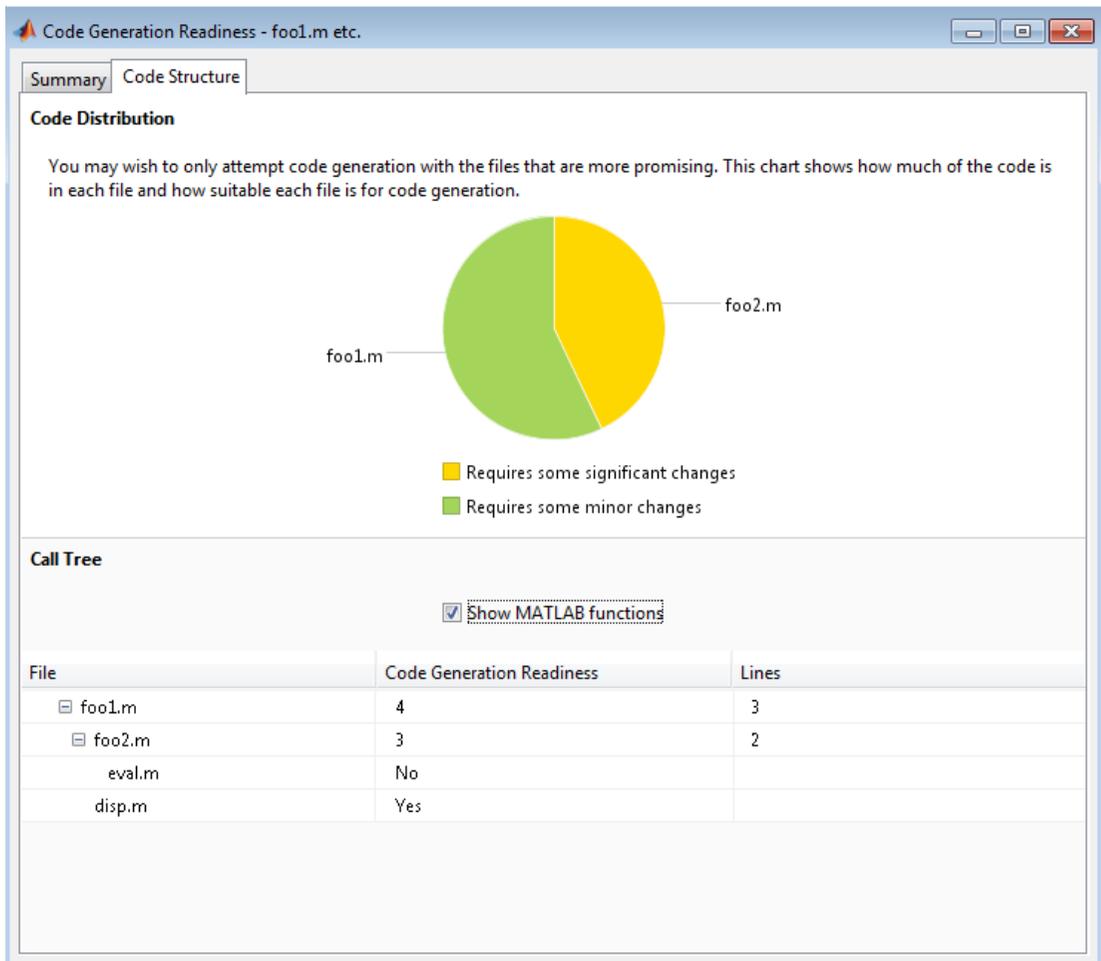
The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function, which code generation does not support. To generate a MEX function for `foo2.m`, modify the code to make the call to `eval` extrinsic.

```
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.

Analyze the MATLAB function myfun that contains a MATLAB table.

```
function outTable = myfun1(A)
outTable = table(A);
end

coder.screener('myfun1');
```

The code generation readiness report indicates that table data types are not supported for code generation.
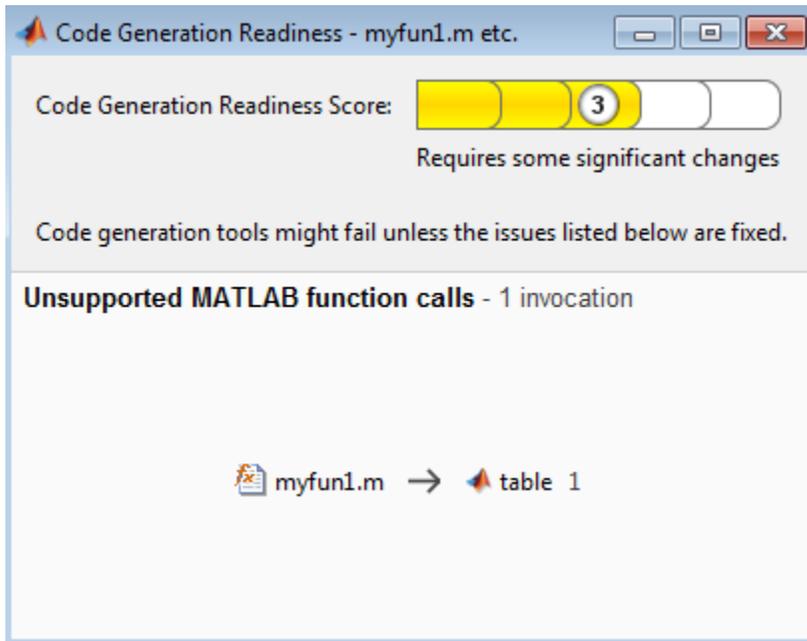
The report assigns `myfun1` a code readiness score of 3. Before generating code, you must fix the reported issues.

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

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 parameterize a MATLAB function

Parameterize a MATLAB function so that it works in MATLAB or 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 the C library for `myabsval.m`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```
`codegen` creates the library `myabsval.lib` and header file `myabsval.h` in the folder /codegen/lib/myabsval. It also 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`.

```matlab
function y = callmyabsval  %#codegen
y = -2.75;
% 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
  % 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
```

Convert `callmyabsval.m` to the MEX function `callmyabsval_mex`.

```matlab
codegen -config:mex callmyabsval codegen/lib/myabsval/myabsval.lib...
     codegen/lib/myabsval/myabsval.h
```

Run the MATLAB function `callmyabsval` .

```matlab
callmyabsval
```

```
ans =

    2.7500
```

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

```matlab
callmyabsval_mex
```

```
ans =
```

```
    2.7500
```

Input Arguments

`target` — code generation target
`'MATLAB'` | `'MEX'` | `'Sfun'` | `'Rtw'` | `'HDL '` | `'Custom'`

Code generation target specified as one of the following values.

`'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')`

See Also

See Also
coder.ceval

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 `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 is applied to bounded dimensions or dimensions that are `1` or `0`, which are 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
```

Tips

*   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

See Also
coder.CellType | coder.ClassType | `codegen` | coder.ArrayType | coder.EnumType | coder.FiType | `coder.newtype` | coder.PrimitiveType | `coder.resize` | coder.StructType | coder.Type

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"

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)
for i = coder.unroll(range)
for i = coder.unroll(range, 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.

`for i = coder.unroll(range)` is a legacy syntax that generates the same code as `coder.unroll()`.

`for i = coder.unroll(range, flag)` is a legacy syntax that generates the same code as `coder.unroll(flag)`.

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.

2-143

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');
```

```matlab
y = zeros(n, 1);
for i = coder.unroll(1:n)
    y(i) = rand();
end
end
```

Use Legacy Syntax to Control `for`-Loop Unrolling

```matlab
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 vargin or vargout in a for-Loop".

See Also

See Also
`coder.inline`

Topics
"Unroll for-Loops"
"Nonconstant Index into vargin or vargout 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 **Target Build Log** 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.

```matlab
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:

```matlab
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 **Target Build Log** 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

Build options, specified as a character vector. 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

Name of user-defined group, specified as a character vector. 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

File name, specified as a character vector. 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.

path — Full path name
character vector

Full path name, specified as a character vector. 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.

Leading Argument	Path Specified by `path`
`'addLinkObjects'`	Path to linkable objects
`'addNonBuildFiles'`	Path to nonbuild-related files
`'addSourceFiles'`, `'addSourcePaths'`	Path to source files

The function adds the path to the end of a path name vector.

`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

Topics
"Customize the Post-Code-Generation Build Process"

Introduced in R2013b

coder.varsize

Package: coder

Declare variable-size array

Syntax

```
coder.varsize('var₁', 'var₂', ...)
coder.varsize('var₁', 'var₂', ..., ubound)
coder.varsize('var₁', 'var₂', ..., ubound, dims)
coder.varsize('var₁', 'var₂', ..., [], dims)
```

Description

`coder.varsize('var₁', 'var₂', ...)` declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each `'varₙ'` is the name of a variable or structure field enclosed in quotes. If the 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. For example, the expression `coder.varsize('data(:).A')` declares that the field A inside each element of `data` is variable sized.

`coder.varsize('var₁', 'var₂', ..., ubound)` declares one or more variables as variable-size data with an explicit upper bound specified in *ubound*. The argument *ubound* must be a constant, integer-valued vector of upper bound sizes for every dimension of each `'varₙ'`. If you specify more than one `'varₙ'`, each variable must have the same number of dimensions.

`coder.varsize('var₁', 'var₂', ..., ubound, dims)` declares one or more variables as variable size with an explicit upper bound and a mix of fixed and varying dimensions specified in *dims*. The argument *dims* is a logical vector, or double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` in *dims* have fixed size; dimensions that correspond to ones or `true` vary in size. If you specify more than one variable, each fixed dimension must have the same value across all `'varₙ'`.

coder.varsize('*var₁*', '*var₂*', ..., [], *dims*) declares one or more variables as variable size with a mix of fixed and varying dimensions. The empty vector [] means that you do not specify an explicit upper bound.

When you do *not* specify *ubound*, the upper bound is computed for each '*varₙ*' in generated code.

When you do *not* specify *dims*, dimensions are assumed to be variable except the singleton ones. A singleton dimension is a dimension for which size(A,dim) = 1.

You must add the coder.varsize declaration before each '*varₙ*' is used (read). You can add the declaration before the first assignment to each '*varₙ*'. However, for a cell array element, the coder.varsize declaration must follow the first assignment to the element. For example:

```
...
x = cell(3, 3);
x{1} = [1 2];
coder.varsize('x{1}');
...
```

You cannot use coder.varsize outside the MATLAB code intended for code generation. For example, the following code does not declare the variable, var, as variable-size data:

```
coder.varsize('var',10);
codegen -config:lib MyFile -args var
```

Instead, include the coder.varsize statement inside MyFile to declare var as variable-size data. Alternatively, you can use coder.typeof to declare var as variable-size outside MyFile. It can then be passed to MyFile during code generation using the -args option. For more information, see coder.typeof.

Examples

Develop a Simple Stack That Varies in Size up to 32 Elements as You Push and Pop Data at Run Time.

Write primary function test_stack to issue commands for pushing data on and popping data from a stack.

```
function test_stack %#codegen
    % The directive %#codegen indicates that the function
```

```
    % is intended for code generation
    stack('init', 32);
    for i = 1 : 20
        stack('push', i);
    end
    for i = 1 : 10
        value = stack('pop');
        % Display popped value
        value
    end
end
```

Write local function `stack` to execute the push and pop commands.

```
function y = stack(command, varargin)
    persistent data;
    if isempty(data)
        data = ones(1,0);
    end
    y = 0;
    switch (command)
    case {'init'}
        coder.varsize('data', [1, varargin{1}], [0 1]);
        data = ones(1,0);
    case {'pop'}
        y = data(1);
        data = data(2:size(data, 2));
    case {'push'}
        data = [varargin{1}, data];
    otherwise
        assert(false, ['Wrong command: ', command]);
    end
end
```

The variable `data` is the stack. The statement `coder.varsize('data', [1, varargin{1}], [0 1])` declares that:

- `data` is a row vector
- Its first dimension has a fixed size
- Its second dimension can grow to an upper bound of 32

Generate a MEX function for `test_stack`:

```
codegen -config:mex test_stack
```

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

Run `test_stack_mex` to get these results:

```
value =
    20

value =
    19

value =
    18

value =
    17

value =
    16

value =
    15

value =
    14

value =
    13

value =
    12

value =
    11
```

At run time, the number of items in the stack grows from zero to 20, and then shrinks to 10.

Declare a Variable-Size Structure Field.

Write a function `struct_example` that declares an array `data`, where each element is a structure that contains a variable-size field:

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

The statement `coder.varsize('data(:).values')` marks as variable-size the field `values` inside each element of the matrix `data`.

Generate a MEX function for `struct_example`:

```
codegen -config:mex struct_example
```

Run `struct_example`.

Each time you run `struct_example` you get a different answer because the function loads the array with random numbers.

Make a Cell Array Variable-Size

Write the function `make_varsz_cell` that defines a local cell array variable `c` whose elements have the same class, but different sizes. Use `coder.varsize` to indicate that `c` has variable-size.

```
function y = make_varsz_cell()
c = {1 [2 3]};
coder.varsize('c', [1 3], [0 1]);
y = c;
end
```

Generate a C static library.

```
codegen -config:lib make_varsz_cell -report
```

In the report, view the MATLAB variables.

`c` is a 1x:3 homogeneous cell array whose elements are 1x:2 double.

Make the Elements of a Cell Array Variable-Size

Write the function `mycell` that defines a local cell array variable `c`. Use `coder.varsize` to make the elements of `c` variable-size.

```
function y =  mycell()
c = {1 2 3};
coder.varsize('c{:}', [1 5], [0 1]);
y = c;
end
```

Generate a C static library.

```
codegen -config:lib mycell -report
```

In the report, view the MATLAB variables.

The elements of `c` are 1-by-:5 arrays of doubles.

Limitations

- If you use the cell function to create a `cell` array, you cannot use `coder.varsize` with that cell array.
- If you use `coder.varsize` with a cell array element, the `coder.varsize` declaration must follow the first assignment to the element. For example:

  ```
  ...
  x = cell(3, 3);
  x{1} = [1 2];
  coder.varsize('x{1}');
  ...
  ```
- You cannot use `coder.varsize` with a cell array input that is heterogeneous.
- You cannot use `coder.varsize` with global variables.
- You cannot use `coder.varsize` with MATLAB class properties.

Tips

- If you use input variables (or 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 use

`coder.varsize` on the array again, unless you also want to specify an upper bound for its size.

- Using `coder.varsize` on an array without explicit upper bounds causes dynamic memory allocation of the array. This dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, use the syntax `coder.varsize('var₁', 'var₂', ..., ubound)` to specify an upper bound for the array size (if you know it in advance).

- A cell array can be variable size only if it is homogeneous. When you use `coder.varsize` with a cell array, the code generator tries to make the cell array homogeneous. It tries to find a class and size that apply to all elements of the cell array. For example, if the first element is double and the second element is 1x2 double, all elements can be represented as 1x:2 double. If the code generator cannot find a common class and size, code generation fails. For example, suppose that the first element of a cell array is char and the second element is double. The code generator cannot find a class that can represent both elements.

See Also

See Also
`codegen` | `size` | `varargin`

Topics
"Code Generation for Variable-Size Arrays"
"Define Variable-Size Structure Fields"
"Defining Variable-Size Global Data"
"Incompatibilities with MATLAB in Variable-Size Support for Code Generation"
"Code Generation for Cell Arrays"

Introduced in R2011a

coder.wref

Pass argument as write-only reference

Syntax

```
coder.wref(arg)
```

Description

coder.wref(arg) passes arg by reference to an external C/C++ function that you call by using coder.ceval. Use coder.wref inside a coder.ceval call only. The code generator assumes that the variable that you pass in coder.wref is write-only. The C/C++ function must write to this variable. If the variable is a vector or an array, the C/C++ function must write to every element of the variable. If the MATLAB code initializes arg before the coder.ceval call, the code generator might remove the initial values. To avoid unpredictable results, the C/C++ function must not read the variable before writing to it.

Use a separate coder.wref construct for each argument that you pass by reference to the function.

Examples

Pass Array by Reference as Write-Only

Pass a variable by reference to a C function that initializes an array.

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 prototype defines array as a pointer because it is passed by reference.

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`.

```
double init_struct(struct MyStruct *my_struct)
{
   my_struct->f1 = 1;
   my_struct->f2 = 2;
}
```

Suppose that a C header file, `MyStruct.h`, defines a structure type named `MyStruct`.

```
...
typedef struct MyStruct
{
    double f1;
    double f2;
```

```
} MyStruct;

double incr_struct(struct MyStruct *my_struct);

...
```

In your MATLAB function, pass a structure as a write-only reference to `init_struct`. To indicate that the structure type for `s` has the name `MyStruct` that is defined in a C header file `MyStruct.h`, use `coder.cstructname`.

```
...
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'MyStruct', 'extern', 'HeaderFile', 'MyStruct.h');
coder.ceval('init_struct', coder.wref(s));
...
```

Pass Structure Field by Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.wef(s.s1.a));
...
```

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 setter or getter method, you cannot pass the property by reference because you cannot access the address of the property. Only the getter or setter method can directly read from or write to this property.

Tips

- 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 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 "External Function Calls from Generated Code".

See Also

See Also
`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.rref`

Topics
"External Function Calls from Generated Code"
"Call External Functions with coder.ceval"

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.

- "Generate Code with Parallel for-Loops (parfor)"

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 `http://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.

See Also

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

See Also
coder.make.BuildConfiguration.getOption |
coder.make.BuildConfiguration.isOption |
coder.make.BuildConfiguration.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

See Also
```
coder.make.BuildConfiguration.addOption |
coder.make.BuildConfiguration.isOption |
coder.make.BuildConfiguration.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 , 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

See Also
```
coder.make.BuildConfiguration.addOption |
coder.make.BuildConfiguration.getOption |
coder.make.BuildConfiguration.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 , 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 , 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

See Also

```
coder.make.BuildConfiguration.addOption |
coder.make.BuildConfiguration.getOption |
coder.make.BuildConfiguration.isOption | coder.make.BuildItem
```

Topics

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 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

See Also
coder.make.BuildItem.getMacro | coder.make.BuildItem.getValue |
coder.make.BuildItem.setMacro | coder.make.BuildItem.setValue

Topics
"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

See Also
coder.make.BuildItem.getMacro | coder.make.BuildItem.getValue |
coder.make.BuildItem.setMacro | coder.make.BuildItem.setValue

Topics
"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

See Also
```
coder.make.BuildItem.getMacro | coder.make.BuildItem.getValue |
coder.make.BuildItem.setMacro | coder.make.BuildItem.setValue
```

Topics
"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

See Also
coder.make.BuildItem.getMacro | coder.make.BuildItem.getValue |
coder.make.BuildItem.setMacro | coder.make.BuildItem.setValue

Topics
"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

See Also
"Properties" on page 3-148 | `coder.make.BuildTool.getDirective` |
`coder.make.BuildTool.setDirective`

Topics
"Example" on page 3-151

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.

2-201

Example: bi

Examples

```
tc = coder.make.ToolchainInfo;
tool = tc.getBuildTool('C Compiler');
blditm = coder.make.BuildItem('CD','.cd')

bldtm =

 Macro  : CD
 Value : .cd

tool.addFileExtension('SourceX',blditm)
value = tool.getFileExtension('SourceX')

value  =

.cd

tool.setFileExtension('SourceX','.ef')
value = tool.getFileExtension('SourceX')

value  =

.ef
```

See Also

See Also
"Properties" on page 3-148 | coder.make.BuildTool.getFileExtension |
coder.make.BuildTool.setFileExtension

Topics

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

See Also
coder.make.BuildTool.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

See Also
"Properties" on page 3-148 | `coder.make.BuildTool.addDirective` |
`coder.make.BuildTool.setDirective`

Topics

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');
blditm = coder.make.BuildItem('CD','.cd')

bldtm =

 Macro  : CD
 Value : .cd

tool.addFileExtension('SourceX',blditm)
value = tool.getFileExtension('SourceX')

value  =

.cd

tool.setFileExtension('SourceX','.ef')
value = tool.getFileExtension('SourceX')

value  =

.ef
```

See Also

See Also
"Properties" on page 3-148 | coder.make.BuildTool.addFileExtension |
coder.make.BuildTool.setFileExtension

Topics

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 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

See Also
```
coder.make.BuildTool.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

See Also
"Properties" on page 3-148 | `coder.make.BuildTool.setPath`

Topics

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 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

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 , 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

See Also
`coder.make.BuildTool.getCommand`

Topics
"Toolchain Definition File with Commentary"

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 , 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

See Also
coder.make.ToolchainInfo.addBuildTool |
coder.make.BuildTool.setCommandPattern

Topics

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 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 installe
% 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',     ['$(MATLAB_ROOT)\extern\lib\' tc.Platform '\micros
tc.addMacro('MW_LIB_DIR',           ['$(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',              '');
```

```matlab
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);
```

```matlab
% ---------------------------------------------
% 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, debu
```

```
cfg.setOption( 'C++ Compiler',          horzcat(cppCompilerOpts, optimsOffOpts, debug
cfg.setOption( 'Linker',                horzcat(linkerOpts,      debugFlag.Linker));
cfg.setOption( 'C++ Linker',             horzcat(linkerOpts,     debugFlag.CppLin
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

See Also
coder.make.BuildTool

Topics

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 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

See Also
"Properties" on page 3-148 | `coder.make.BuildTool.addDirective` |
`coder.make.BuildTool.getDirective`

Topics

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 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');
blditm = coder.make.BuildItem('CD','.cd')

bldtm =

 Macro  : CD
 Value : .cd

tool.addFileExtension('SourceX',blditm)
value = tool.getFileExtension('SourceX')

value  =

.cd

tool.setFileExtension('SourceX','.ef')
value = tool.getFileExtension('SourceX')

value  =
```

2-231

```
.ef
```

See Also

See Also
"Properties" on page 3-148 | `coder.make.BuildTool.addFileExtension` |
`coder.make.BuildTool.getFileExtension`

Topics

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 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 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"

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 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

See Also
"Properties" on page 3-148 | `coder.make.BuildTool.getPath`

Topics

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 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

See Also
`coder.make.ToolchainInfo.validate`

Topics
"Troubleshooting Custom Toolchain Validation"

addAttribute

Class: coder.make.ToolchainInfo
Package: coder.make

Add custom attribute to `Attributes`

Syntax

```
h.addAttribute(att_name, att_value)
h.addAttribute(att_name)
```

Description

`h.addAttribute(att_name, att_value)` adds a custom attribute with the specified name to `coder.make.ToolchainInfo.Attributes`. Use `att_value` to override the default attribute value, `true`.

`h.addAttribute(att_name)` adds an attribute and initializes its value to `true`.

All attributes are optional. You can add attributes for the toolchain to use. The following attributes are used during the build process:

- `TransformPathsWithSpaces` (value = True/False) When enabled, this method looks 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, this returns the '~' version on Windows. On UNIX platforms, paths with spaces are returned with the spaces escaped.

- `RequiresBatchFile`: (value = True/False) When enabled, creates a batch file that runs the make file that is generated.

- `SupportsUNCPaths`: (value = True/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`: (not defined or value = True) Wraps each path in double quotes if they contain spaces.
- `RequiresCommandFile`: (value = True/False) This is used to handle long archiver/ linker lines in Windows. If specified, the make file replaces a long object list with a call to a command file.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo 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

Name of attribute, specified as a character vector.

Data Types: `char`

att_value — Value of attribute
true (default)

Attribute value. Any data type.

Examples

Add an attribute and initialize its value, overriding the default value

```
h.Attribute

ans =


# ------------------
# "Attribute" List
# ------------------
(empty)
```

2-243

```
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 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

See Also
```
coder.make.ToolchainInfo.addAttribute |
coder.make.ToolchainInfo.getAttribute |
coder.make.ToolchainInfo.getAttributes
| coder.make.ToolchainInfo.isAttribute |
coder.make.ToolchainInfo.removeAttribute
```

Topics
"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.

`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. The build configuration must have a name that is unique within coder.make.ToolchainInfo.BuildConfigurations.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo 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

Build configuration name, specified as a character vector.

Data Types: char

bldcfg_handle — BuildConfiguration object handle
character vector

Handle of coder.make.BuildConfiguration object

bldcfg_desc — Build configuration description
character vector

Build configuration description, specified as a character vector.

Data Types: char

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

See Also
coder.make.BuildConfiguration | coder.make.BuildItem |
coder.make.BuildTool

Topics
"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 3-151 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 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

Build tool name, specified as a character vector.

Data Types: `char`

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

Output Arguments

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

Examples

Refer to the coder.make.BuildTool "Example" on page 3-151 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

See Also
`coder.make.BuildTool` | `coder.make.ToolchainInfo.addBuildTool`
| `coder.make.ToolchainInfo.getBuildTool` |
`coder.make.ToolchainInfo.removeBuildTool` |
`coder.make.ToolchainInfo.setBuildTool`

Topics
"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 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

Intrinsic macro name or names, specified as a character vector or cell array of character
vectors.

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

See Also
coder.make.ToolchainInfo.addMacro | coder.make.ToolchainInfo.getMacro
| coder.make.ToolchainInfo.removeMacro
| coder.make.ToolchainInfo.setMacro |
coder.make.ToolchainInfo.addIntrinsicMacros |
coder.make.ToolchainInfo.removeIntrinsicMacros

Topics
"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 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

Name of macro.

macrovalue — Value of macro
character vector | cell

Value of the macro, specified as a character vector or cell array.

If the value contains MATLAB functions or other macros, ToolchainInfo interprets the value of functions and macros.

Data Types: `cell` | `char`

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

See Also
`coder.make.ToolchainInfo.addMacro` | `coder.make.ToolchainInfo.getMacro` | `coder.make.ToolchainInfo.removeMacro` | `coder.make.ToolchainInfo.setMacro` | `coder.make.ToolchainInfo.addIntrinsicMacros` | `coder.make.ToolchainInfo.removeIntrinsicMacros`

Topics
"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 3-151 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 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

Build tool name, specified as a character vector.

Data Types: char

bldtl_handle — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

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

See Also
```
coder.make.ToolchainInfo.addPostbuildTool |
coder.make.ToolchainInfo.getPostbuildTool |
coder.make.ToolchainInfo.removePostbuildTool
| coder.make.ToolchainInfo.setPostbuildTool |
coder.make.ToolchainInfo.addPostDownloadTool |
coder.make.ToolchainInfo.addPostExecuteTool
```

Topics
"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 3-151 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 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

Build tool name, specified as a character vector.

Data Types: `char`

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

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

See Also
```
coder.make.ToolchainInfo.addPostbuildTool |
coder.make.ToolchainInfo.getPostbuildTool |
coder.make.ToolchainInfo.removePostbuildTool
| coder.make.ToolchainInfo.setPostbuildTool |
coder.make.ToolchainInfo.addPostDownloadTool |
coder.make.ToolchainInfo.addPostExecuteTool
```

Topics
"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 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

Name of post execute tool, specified as a character vector.

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

Examples

Refer to the coder.make.BuildTool "Example" on page 3-151 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

See Also
```
coder.make.ToolchainInfo.addPostbuildTool |
coder.make.ToolchainInfo.addPostDownloadTool
```

Topics

"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 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

Build tool name, specified as a character vector.

Data Types: `char`

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

See Also

See Also
```
coder.make.ToolchainInfo.addPrebuildTool |
coder.make.ToolchainInfo.getPrebuildTool |
coder.make.ToolchainInfo.removePrebuildTool |
coder.make.ToolchainInfo.setPrebuildTool
```

Topics
"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 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

See Also
`coder.make.ToolchainInfo.setup` | `coder.make.ToolchainInfo.validate`

Topics
"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 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

Name of attribute, specified as a character vector.

Data Types: `char`

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

See Also
```
coder.make.ToolchainInfo.addAttribute |
coder.make.ToolchainInfo.getAttribute |
coder.make.ToolchainInfo.getAttributes
| coder.make.ToolchainInfo.isAttribute |
coder.make.ToolchainInfo.removeAttribute
```

Topics
"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 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

See Also
```
coder.make.ToolchainInfo.addAttribute |
coder.make.ToolchainInfo.getAttribute |
coder.make.ToolchainInfo.getAttributes
| coder.make.ToolchainInfo.isAttribute |
coder.make.ToolchainInfo.removeAttribute
```

Topics
"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 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

Build configuration name, specified as a character vector.

Data Types: `char`

Output Arguments

bldcfg_handle — BuildConfiguration object handle
character vector

2-273

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

See Also
```
coder.make.ToolchainInfo.getBuildConfiguration |
coder.make.ToolchainInfo.removeBuildConfiguration
| coder.make.ToolchainInfo.setBuildConfiguration |
coder.make.ToolchainInfo.setBuildConfigurationOption
```

Topics
"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 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

Build tool name, specified as a character vector.

Data Types: `char`

Output Arguments

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

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

See Also
```
coder.make.BuildTool | coder.make.ToolchainInfo.addBuildTool
| coder.make.ToolchainInfo.getBuildTool |
coder.make.ToolchainInfo.removeBuildTool |
coder.make.ToolchainInfo.setBuildTool
```

Topics
"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 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

Name of macro.

Output Arguments

`macrovalue` — Value of macro
character vector | cell

Value of the macro, specified as a character vector or cell array.

If the value contains MATLAB functions or other macros, ToolchainInfo interprets the value of functions and macros.

Data Types: cell | char

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')


ans  =

C:\cygwin\bin\


h.removeMacro('CYGWIN')
```

See Also

See Also

coder.make.ToolchainInfo.addMacro | coder.make.ToolchainInfo.getMacro | coder.make.ToolchainInfo.removeMacro | coder.make.ToolchainInfo.setMacro | coder.make.ToolchainInfo.addIntrinsicMacros | coder.make.ToolchainInfo.removeIntrinsicMacros

Topics

"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 assigns it to a handle.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo 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

Build tool name, specified as a character vector.

Data Types: `char`

Output Arguments

bldtl_handle — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

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

See Also
```
coder.make.ToolchainInfo.addPostbuildTool |
coder.make.ToolchainInfo.getPostbuildTool |
coder.make.ToolchainInfo.removePostbuildTool
```

```
| coder.make.ToolchainInfo.setPostbuildTool |
coder.make.ToolchainInfo.addPostDownloadTool |
coder.make.ToolchainInfo.addPostExecuteTool
```

Topics
"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 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

Build tool name, specified as a character vector.

Data Types: `char`

Output Arguments

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

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

See Also
```
coder.make.ToolchainInfo.addPrebuildTool |
coder.make.ToolchainInfo.getPrebuildTool |
```

```
coder.make.ToolchainInfo.removePrebuildTool |
coder.make.ToolchainInfo.setPrebuildTool
```

Topics
"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 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) in the MATLAB Object-Oriented Programming documentation.

Examples

```
ans = h.getSupportedLanguages


ans =

    'Asm/C'    'Asm/C++'    'Asm/C/C++'    'C'    'C++'    'C/C++'
```

See Also

See Also
```
coder.make.ToolchainInfo
```

Topics
"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 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

Name of attribute, specified as a character vector.

Data Types: `char`

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

See Also

```
coder.make.ToolchainInfo.addAttribute |
coder.make.ToolchainInfo.getAttribute |
coder.make.ToolchainInfo.getAttributes
| coder.make.ToolchainInfo.isAttribute |
coder.make.ToolchainInfo.removeAttribute
```

Topics

"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 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

Name of attribute, specified as a character vector.

Data Types: `char`

Examples

```
h.addAttribute('FirstAttribute')
h.isAttribute('FirstAttribute')
```

```
ans =

     1

h.removeAttribute('FirstAttribute')
h.isAttribute('FirstAttribute')


ans =

     0
```

See Also

See Also
```
coder.make.ToolchainInfo.addAttribute |
coder.make.ToolchainInfo.getAttribute |
coder.make.ToolchainInfo.getAttributes
| coder.make.ToolchainInfo.isAttribute |
coder.make.ToolchainInfo.removeAttribute
```

Topics
"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 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

Build configuration name, specified as a character vector.

Data Types: char

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

See Also
```
coder.make.ToolchainInfo.getBuildConfiguration |
coder.make.ToolchainInfo.removeBuildConfiguration
| coder.make.ToolchainInfo.setBuildConfiguration |
coder.make.ToolchainInfo.setBuildConfigurationOption
```

Topics
"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 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

Build tool name, specified as a character vector.

Data Types: `char`

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

See Also
```
coder.make.BuildTool | coder.make.ToolchainInfo.addBuildTool
| coder.make.ToolchainInfo.getBuildTool |
coder.make.ToolchainInfo.removeBuildTool |
coder.make.ToolchainInfo.setBuildTool
```

Topics
"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 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

Intrinsic macro name or names, specified as a character vector or cell array of character vectors.

Examples

```
h.addIntrinsicMacros('GCCROOT')
h.getMacro('GCCROOT')
```

```
ans =

    []

h.removeIntrinsicMacros('GCCROOT')
h.getMacro('GCCROOT')
```

See Also

See Also
coder.make.ToolchainInfo.addMacro | coder.make.ToolchainInfo.getMacro
| coder.make.ToolchainInfo.removeMacro
| coder.make.ToolchainInfo.setMacro |
coder.make.ToolchainInfo.addIntrinsicMacros |
coder.make.ToolchainInfo.removeIntrinsicMacros

Topics
"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 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

Name of macro.

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')


ans  =
```

```
C:\cygwin\bin\
```

```
h.removeMacro('CYGWIN')
```

See Also

See Also
```
coder.make.ToolchainInfo.addMacro | coder.make.ToolchainInfo.getMacro
| coder.make.ToolchainInfo.removeMacro
| coder.make.ToolchainInfo.setMacro |
coder.make.ToolchainInfo.addIntrinsicMacros |
coder.make.ToolchainInfo.removeIntrinsicMacros
```

Topics
"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 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

Build tool name, specified as a character vector.

Data Types: `char`

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

See Also
```
coder.make.ToolchainInfo.addPostbuildTool |
coder.make.ToolchainInfo.getPostbuildTool |
coder.make.ToolchainInfo.removePostbuildTool
| coder.make.ToolchainInfo.setPostbuildTool |
coder.make.ToolchainInfo.addPostDownloadTool |
coder.make.ToolchainInfo.addPostExecuteTool
```

Topics
"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 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

Build tool name, specified as a character vector.

Data Types: `char`

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

See Also
```
coder.make.ToolchainInfo.addPrebuildTool |
coder.make.ToolchainInfo.getPrebuildTool |
coder.make.ToolchainInfo.removePrebuildTool |
coder.make.ToolchainInfo.setPrebuildTool
```

Topics
"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 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

Build configuration name, specified as a character vector.

Data Types: char

`bldcfg_handle` — BuildConfiguration object handle
character vector

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

See Also
```
coder.make.ToolchainInfo.getBuildConfiguration |
coder.make.ToolchainInfo.removeBuildConfiguration
| coder.make.ToolchainInfo.setBuildConfiguration |
coder.make.ToolchainInfo.setBuildConfigurationOption
```

Topics
"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, options)
```

Description

`h.setBuildConfigurationOption(buildconfignames, 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 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

Build configuration name or `'all'`, specified as a character vector.

buildobjectname — BuildTool object name
character vector

`BuildTool` object name, specified as a character vector.

options — Build configuration options
character vector

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

See Also
```
coder.make.ToolchainInfo.getBuildConfiguration |
coder.make.ToolchainInfo.removeBuildConfiguration
| coder.make.ToolchainInfo.setBuildConfiguration |
coder.make.ToolchainInfo.setBuildConfigurationOption
```

Topics
"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 object, specified using an object handle, such as `h`. To create
`h`, enter `h = coder.make.ToolchainInfo` in a MATLAB Command Window.

`platform` — Host computer platform
character vector

Host computer platform, specified as a scalar. The values can be:

- `WIN32`
- `WIN64`
- `MACI64`
- `GLNXA64`

Data Types: `char`

Examples

The `intel_tc.m` file from , 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 computer running Windows and the value of `coder.make.ToolchainInfo.BuildArtifact` is `gmake makefile`.

See Also

Topics

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 3-151 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 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

Build tool name, specified as a character vector.

Data Types: `char`

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('examplename')
h.setBuildTool('Archiver',bt)
```

See Also

See Also
```
coder.make.BuildTool | coder.make.ToolchainInfo.addBuildTool
| coder.make.ToolchainInfo.getBuildTool |
coder.make.ToolchainInfo.removeBuildTool |
coder.make.ToolchainInfo.setBuildTool
```

Topics
"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 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

Name of macro.

`macrovalue` — Value of macro
character vector | cell

Value of the macro, specified as a character vector or cell array.

If the value contains MATLAB functions or other macros, ToolchainInfo interprets the value of functions and macros.

Data Types: `cell` | `char`

Examples

```
h.setMacro('CYGWIN','C:\cygwin\');
h.getMacro('CYGWIN')


ans =

C:\cygwin\bin\


h.removeMacro('CYGWIN')
```

See Also

See Also
```
coder.make.ToolchainInfo.addMacro | coder.make.ToolchainInfo.getMacro
| coder.make.ToolchainInfo.removeMacro
| coder.make.ToolchainInfo.setMacro |
coder.make.ToolchainInfo.addIntrinsicMacros |
coder.make.ToolchainInfo.removeIntrinsicMacros
```

Topics
"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.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo 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

Build tool name, specified as a character vector.

Data Types: `char`

`bldtl_handle` — BuildTool object handle
character vector

Handle of `coder.make.BuildTool` object.

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('examplename')
h.addPostbuildTool('toolname')
h.setPostbuildTool('toolname',bt)
```

See Also

See Also
```
coder.make.ToolchainInfo.addPostbuildTool |
coder.make.ToolchainInfo.getPostbuildTool |
coder.make.ToolchainInfo.removePostbuildTool
| coder.make.ToolchainInfo.setPostbuildTool |
coder.make.ToolchainInfo.addPostDownloadTool |
coder.make.ToolchainInfo.addPostExecuteTool
```

Topics
"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.

Input Arguments

h — ToolchainInfo object handle
variable

A coder.make.ToolchainInfo 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

Build tool name, specified as a character vector.

Data Types: `char`

Examples

```
h = coder.make.ToolchainInfo;
bt = coder.make.BuildTool('examplename');
h.addPrebuildTool('toolname');
```

```
h.setPrebuildTool('toolname',bt)
```

See Also

See Also
```
coder.make.ToolchainInfo.addPrebuildTool |
coder.make.ToolchainInfo.getPrebuildTool |
coder.make.ToolchainInfo.removePrebuildTool |
coder.make.ToolchainInfo.setPrebuildTool
```

Topics
"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 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

See Also
coder.make.ToolchainInfo.cleanup | coder.make.ToolchainInfo.validate

Topics
"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 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

Evaluates `coder.make.ToolchainInfo.setup` for the toolchain.

`'cleanup'` — Cleanup argument for validate operation.
character vector

Evaluates `coder.make.ToolchainInfo.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

See Also
coder.make.ToolchainInfo.cleanup | coder.make.ToolchainInfo.setup

Topics
"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

See Also
```
coder.make.getToolchainInfoFromRegistry |
coder.make.BuildTool | coder.make.ToolchainInfo.addBuildTool
| coder.make.ToolchainInfo.getBuildTool |
coder.make.ToolchainInfo.removeBuildTool |
coder.make.ToolchainInfo.setBuildTool
```

Topics
"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 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

See Also
```
coder.make.getDefaultToolchain | coder.make.BuildTool
| coder.make.ToolchainInfo.addBuildTool
| coder.make.ToolchainInfo.getBuildTool |
coder.make.ToolchainInfo.removeBuildTool |
coder.make.ToolchainInfo.setBuildTool
```

Topics
"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

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 vectorcan 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

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

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

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

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

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

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

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 2-350. 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 2-350

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

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 2-352 in the current "build context" on page 2-352 . 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 2-352 in the current "build context" on page 2-352, 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 2-352.

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

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 2-355.

You must implement this method in a subclass of `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 2-355.

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

Topics
"Build Information Object"
"Build Information Methods"

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

See Also
coder.FixptConfig | coder.FixptConfig.hasDesignRangeSpecification
| coder.FixptConfig.removeDesignRangeSpecification
| coder.FixptConfig.clearDesignRangeSpecifications |
coder.FixptConfig.getDesignRangeSpecification | codegen

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

See Also
codegen | coder.config | coder.FixptConfig

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

```matlab
% 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

See Also
coder.FixptConfig | coder.FixptConfig.addDesignRangeSpecification
| coder.FixptConfig.removeDesignRangeSpecification

| coder.FixptConfig.hasDesignRangeSpecification | coder.FixptConfig.getDesignRangeSpecification | codegen

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

See Also
coder.FixptConfig | coder.FixptConfig.addDesignRangeSpecification
| coder.FixptConfig.hasDesignRangeSpecification |
coder.FixptConfig.removeDesignRangeSpecification |
coder.FixptConfig.clearDesignRangeSpecifications | codegen

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

See Also
coder.FixptConfig | coder.FixptConfig.addDesignRangeSpecification
| coder.FixptConfig.removeDesignRangeSpecification
| coder.FixptConfig.clearDesignRangeSpecifications |
coder.FixptConfig.getDesignRangeSpecification | `codegen`

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

See Also
coder.FixptConfig | coder.FixptConfig.addDesignRangeSpecification
| coder.FixptConfig.clearDesignRangeSpecifications
| coder.FixptConfig.hasDesignRangeSpecification |
coder.FixptConfig.getDesignRangeSpecification | codegen

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.

- "Replace the exp Function with a Lookup Table"
- "Replace a Custom Function with a Lookup Table"

See Also

See Also
codegen | `coder.config` | coder.FixptConfig

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

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'`.

```matlab
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
```

- "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)

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');
```

- "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)

See Also

See Also
coder.ExternalDependency | coder.BuildConfig

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

Class Reference

coder.ArrayType class

Package: coder
Superclasses: coder.Type

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) in the MATLAB documentation.

See Also

coder.EnumType | coder.FiType | coder.PrimitiveType | coder.StructType | coder.CellType | `coder.resize` | coder.Type | `coder.newtype` | `coder.typeof` | `codegen`

Topics

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

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

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(ctx)
            if  ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, ctx)
            [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

            % Header files
            hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
```

3-5

```matlab
        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
        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
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
        c = coder.ceval('adder', a, b);


        % Call the terminate function after
        % calling adder for the last time.

        coder.ceval('adder_terminate');
    end
end
```

```
    end
end
```

See Also

See Also
coder.ExternalDependency | coder.HardwareImplementation |
coder.make.ToolchainInfo | coder.target

Introduced in R2013b

coder.CellType class

Package: coder
Superclasses: coder.ArrayType

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

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
character vector

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.

Must be a nonempty character vector

`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

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

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

See Also
coder.ArrayType | coder.Constant | coder.EnumType | coder.FiType |
coder.PrimitiveType | coder.StructType | coder.Type | `codegen` | `coder.newtype` |
`coder.resize` | `coder.typeof`

Topics
"Code Generation for Cell Arrays"

Introduced in R2015b

coder.ClassType class

Package: coder
Superclasses: coder.ArrayType

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. Enclose the name in single quotes. For example:

```
t = coder.newtype('myValueClass');
```

Properties

When you create a `coder.ClassType` object by using `coder.typeof`, the `coder.ClassType` object has the same properties as the object from which it is constructed.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

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

coder.EnumType | coder.EnumType | coder.FiType | coder.Constant | coder.ArrayType | `coder.typeof` | `coder` | `coder.cstructname` | coder.Type | coder.PrimitiveType | `coder.newtype` | `coder.resize` | `codegen`

Topics

Introduced in R2017a

coder.CodeConfig class

Package: coder

codegen configuration object

Description

A coder.CodeConfig object contains the configuration parameters that the codegen function requires to generate standalone C/C++ libraries and executables. Use the -config option to pass this object to the codegen function.

Construction

cfg = coder.config('lib') creates a code generation configuration object for C/C++ static library generation. If the Embedded Coder product is not installed, it creates a coder.CodeConfig object. Otherwise, it creates a coder.EmbeddedCodeConfig object.

cfg = coder.config('dll') creates a code generation configuration object for C/C++ dynamic library generation. If the Embedded Coder product is not installed, it creates a coder.CodeConfig object. Otherwise, it creates a coder.EmbeddedCodeConfig object.

cfg = coder.config('exe') creates a code generation configuration object for C/C++ executable generation. If the Embedded Coder product is not installed, it creates a coder.CodeConfig object. Otherwise, it creates a coder.EmbeddedCodeConfig object.

cfg = coder.config(*output_type*, 'ecoder', false) creates a coder.CodeConfig object for the specified output type even if the Embedded Coder product is installed.

cfg = coder.config(*output_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 an Embedded Coder license is available.

Properties

BuildConfiguration

Specify build configuration. `'Faster Builds'`, `'Faster Runs'`, `'Debug'`, `'Specify'`.

Default: `'Faster Builds'`

CodeReplacementLibrary

Specify a code replacement library for the generated code.

Value	Description
`'None'`	Does not use a code replacement library.
`'GNU C99 extensions'`	Generates calls to the GNU® gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
`'Intel IPP for x86-64 (Windows)'`	Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.
`'Intel IPP/SSE for x86-64 (Windows)'`	Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
`'Intel IPP for x86-64 (Windows using MinGW compiler)'`	Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.
`'Intel IPP/SSE for x86-64 (Windows using MinGW compiler)'`	Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
`'Intel IPP for x86/Pentium (Windows)'`	Generates calls to the IPP library for the x86/Pentium Windows platform.
`'Intel IPP/SSE x86/Pentium (Windows)'`	Generates calls to the IPP and SSE libraries for the x86/Pentium Windows platform.
`'Intel IPP for x86-64 (Linux)'`	Generates calls to the IPP library for the x86-64 Linux platform.

Value	Description
`'Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)'`	Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

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.

Default: `'None'`

CompileTimeRecursionLimit

For compile-time recursion, control the number of copies of a function that are allowed in the generated code. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to 0. The default compile-time recursion limit is high 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".

Default: *integer*, 50

ConstantFoldingTimeout

Specify the maximum number of instructions that the constant folder will execute before stopping. In some situations, code generation might require specific instructions to be constant. Increase this value if code generation is failing.

Default: *integer*, 10000

CustomHeaderCode

Specify code to appear near the top of each C/C++ header file generated from your MATLAB algorithm code. Specify code as a character vector.

Default: ''

CustomInclude

Specify a space-separated list of include folders to add to the include path when compiling the generated code. Specify list of include folders as a character vector.

If your list includes Windows paths that contain spaces, enclose each instance in double quotes, for example:

'C:\Project "C:\Custom Files"'

Default: ''

CustomInitializer

Specify code to appear in the initialize function of the generated .c or .cpp file. Specify code as a character vector.

Default: ''

CustomLAPACKCallback

Specify the name of a LAPACK callback class that derives from coder.LAPACKCallback. Specify name 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.

Default: `''`

CustomLibrary

Specify a space-separated list of static library or object files to link with the generated code. Specify list as a character vector.

Default: `''`

CustomSource

Specify a space-separated list of source files to compile and link with the generated code. Specify the list of source files as a character vector.

The build process looks 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 intended. For example, suppose that you specify `foo.cpp` as a source file. If `foo.c` and `foo.cpp` are in folders on the search path, you cannot be sure whether the build process uses `foo.c` or `foo.cpp`.

Default: `''`

CustomSourceCode

Specify code to appear near the top of the generated `.c` or `.cpp` file, outside of a function. Specify code as a character vector.

Do not specify a C static function definition.

Default: `''`

CustomTerminator

Specify code to appear in the terminate function of the generated `.c` or `.cpp` file. Specify code as a character vector.

Default: *character vector*, `''`

CustomToolchainOptions

Specify custom settings for each tool in the selected toolchain, as a cell array.

Dependencies:

- `Toolchain` determines which tools and options appear in the cell.
- Setting `BuildConfiguration` to `Specify` enables the options specified by `CustomToolchainOptions`.

Start by getting the current options and values. For example:

```
rtwdemo_sil_topmodel;
set_param(gcs, 'BuildConfiguration', 'Specify')
opt = get_param(gcs, '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".

Default: *cell array*

DataTypeReplacement

Specify whether to use built-in C data types or pre-defined types from `rtwtypes.h` in generated code.

Set to `'CoderTypeDefs'` to use the data types from `rtwtypes.h`. Otherwise, to use built-in C data types, retain default value or set to `'CBuiltIn'`.

Default: `'CBuiltIn'`

Description

Description of the `coder.CodeConfig` object specified as a character vector.

Default: `'class CodeConfig: C code generation configuration.'`

DynamicMemoryAllocation

Control use of dynamic memory allocation for variable-size data.

By default, dynamic memory allocation is enabled for variable-size arrays whose size (in bytes) is greater than or equal to `DynamicMemoryAllocationThreshold`. `codegen` allocates memory for this variable-size data dynamically on the heap.

Set this property to `'Off'` to allocate memory statically on the stack. Set it to `'AllVariableSizeArrays'` to allocate memory for all variable-size arrays dynamically on the heap. You **must** use dynamic memory allocation for unbounded variable-size data.

Dependencies:

- `EnableVariableSizing` enables this parameter.
- Setting this parameter to `'Threshold'` enables the `DynamicMemoryAllocationThreshold` parameter.

Default: `'Threshold'`

DynamicMemoryAllocationThreshold

Specify the size threshold in bytes. `codegen` allocates memory on the heap for variable-size arrays whose size is greater than or equal to this threshold.

Dependency:

- Setting `DynamicMemoryAllocation` to `'Threshold'` enables this parameter.

Default: 65536

EnableAutoExtrinsicCalls

Specify whether MATLAB Coder must treat common visualization functions as extrinsic functions. When this option is enabled, MATLAB Coder detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. For MEX code generation, MATLAB Coder calls out to MATLAB for these functions. For standalone code generation, MATLAB Coder does not generate code for these visualization functions. This capability reduces the amount of time that you spend making your code suitable for code generation. It also removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

Default: `true`

EnableMemcpy

Enable the `memcpy` optimization. To optimize code that copies consecutive array elements, the `memcpy` optimization replaces the code with a `memcpy` call. A `memcpy` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element

assignments. The code generatorinvokes the `memcpy` optimization when the following conditions are true:

- `EnableMemcpy` is `true`.
- The number of bytes to copy is greater than or equal to `MemcpyThreshold`. The number of bytes to copy is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

See "memcpy Optimization".

Default: `true`

EnableOpenMP

If possible, enable OpenMP. Using the OpenMP library, the C/C++ code that MATLAB Coder generates for `parfor`-loops can run on multiple threads. With OpenMP disabled, MATLAB Coder treats `parfor`-loops as for-loops and generates C/C++ code that runs on a single thread.

Default: `true`

EnableRuntimeRecursion

Allow recursive functions in generated code. If your MATLAB code requires run-time recursion and this parameter is `false`, code generation fails.

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`.

Default: `true`

EnableVariableSizing

Enable support for variable-size arrays.

Dependency:

- This parameter enables the parameter `DynamicMemoryAllocation`.

Default: `true`

`FilePartitionMethod`

Specify whether to generate separate C/C++ files for each MATLAB language file (`'MapMFileToCFile'`) or generate a single file for C/C++ functions that map to your MATLAB entry-point functions (`'SingleFile'`). When you choose `'SingleFile'`, the code generator creates separate C/C++ files for utility functions.

Default: `'MapMFileToCFile'`

`GenCodeOnly`

Specify code generation versus an executable or static library build.

Default: `false`

`GenerateComments`

Place comments in the generated files.

Dependency:

•   Enables `MATLABSourceComments`.

Default: `true`

`GenerateExampleMain`

Specify whether to generate an example C/C++ main function. If example main generation is enabled, MATLAB Coder generates source and header files for the main function in the `examples` subfolder of the build folder. For C code generation, it generates the files `main.c` and `main.h`. For C++ code generation, it generates the files `main.cpp` and `main.h`.

The example main function declares and initializes data. It calls entry-point functions but does not use values returned from the entry-point functions.

Before you use the example main files, copy them to another location and modify them to meet the requirements of your application.

Value	Description
`'DoNotGenerate'`	Does not generate an example C/C++ main function.

Value	Description
`'GenerateCodeOnly'`	Generates an example C/C++ main function but does not compile it.
`'GenerateCodeAndCompile'`	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`, MATLAB Coder does not compile the C/C++ main function.

Default: `'GenerateCodeOnly'`

GenerateMakefile

Specify whether to generate a makefile during the build process.

Default: `true`

GenerateReport

Document generated code in an HTML report.

Default: `false`

HardwareImplementation

Handle to `coder.HardwareImplementation` object that specifies hardware-specific configuration parameters for C/C++ code generation. Defaults to a `coder.HardwareImplementation` object with properties that are set for the MATLAB host computer.

InitFltsAndDblsToZero

Enable the `memset` optimization for assignment of float or double 0 to consecutive array elements. To optimize code that assigns a constant value to consecutive array elements, the `memset` optimization replaces the code with a `memset` call. A `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments. To assign float or double 0 to consecutive array elements, the code generatorinvokes the `memset` optimization when the following conditions are true:

- `InitFltsAndDblsToZero` is `true`.
- The number of bytes to assign is greater than or equal to `MemcpyThreshold`. The number of bytes to assign is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

See "memset Optimization".

Default: `true`

InlineStackLimit

Specify the stack size limit on inlined functions. This specification determines the amount of stack space allocated for local variables of the inlined function.

Specifying a limit on 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 feature is especially important for embedded processors, where stack size can be limited.

Default: 4000

InlineThreshold

Specify function size for inline threshold. Unless there are conflicts with other inlining conditions, MATLAB Coder inlines functions that are smaller than this size.

The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. You must experiment with this parameter to obtain the inlining behavior that you want. For instance, if the default setting for this parameter is leading to large functions being inlined and in turn generating large C code, you can tune the parameter in steps until you are satisfied with the size of generated code.

Default: 10

InlineThresholdMax

Specify the maximum size of functions after inlining. If the size of the calling function after inlining exceeds `InlineThresholdMax`,MATLAB Coder does not inline the called function.

The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. You must experiment with this parameter to obtain the inlining behavior that you want. For instance, if the default setting for this parameter is leading to large functions being inlined and in turn generating large C code, you can tune the parameter in steps until you are satisfied with the size of generated code.

Default: 200

LaunchReport

Specify whether to display a report after code generation is complete or an error occurs.

Default: true

MATLABSourceComments

Include MATLAB source code as comments in the generated code.

Dependency:

• GenerateComments enables this parameter.

Default: false

MaxIdLength

Specify maximum number of characters in 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 parameter 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.

Minimum is 31. Maximum is 256.

Default: 31

MemcpyThreshold

Specify the minimum number of bytes for the code generatorto invoke the memcpy optimization or the memset optimization. To optimize generated code that copies

consecutive array elements, the code generatortries to replace the code with a `memcpy` call. To optimize generated code that assigns a literal constant to consecutive array elements, the code generatortries to replace the code with a `memset` call. A `memcpy` or `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

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.

See "memcpy Optimization" and "memset Optimization".

Default: 64

MultiInstanceCode

Generate reusable, multi-instance code that is reentrant.

Default: `false`

Name

Name of the configuration object.

Default: *character vector*, `'CodeConfig'`

OutputType

Specify whether to generate a standalone C/C++ static library, dynamic library, or executable. Set to `'LIB'` to generate a static library, `'DLL'` to generate a dynamic library, or `'EXE'` to generate an executable.

Default: `'LIB'`

PassStructByReference

Specify whether to pass structures by reference to entry-point functions. Set to `true` to pass structures by reference, which reduces memory usage and execution time by minimizing the number of copies of parameters at entry-point function boundaries. Set to `false` to pass structures by value.

This parameter applies only to entry-point functions.

If you set this parameter to `true`, an entry-point function that writes to a field of a structure parameter overwrites the input value.

Default: true

PostCodeGenCommand

Specify command to customize build processing after code generation using codegen.
Specify command as a character vector.

Default: ''

PreserveVariableNames

Specify which variable names the code generator must preserve in the generated code.

Value	Description
'None'	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.

Value	Description
`'All'`	Preserve all variable names. This parameter value disables variable reuse. Use it only for testing or debugging, not for production code.

Default: `'None'`

ReportPotentialDifferences

Specify whether to report potential behavior differences between generated code and MATLAB code. If `ReportPotentialDifferences` is `true`, the code generation report has a tab that lists the potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

Default: `true`

ReservedNameArray

Enter a space-separated list of names that MATLAB Coder is not to use for naming functions or variables. Specify list as a character vector.

Default: `''`

RuntimeChecks

Enable run-time error detection and reporting in the generated C/C++ code. If you select this option, 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.

Default: `false`

SaturateOnIntegerOverflow

Overflows saturate to either the minimum or maximum value that the data type can represent. Otherwise, the overflow behavior depends on your target C compiler. 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.

Default: `true`

StackUsageMax

Specify the maximum stack usage per application in bytes. Set a limit that is lower than the available stack size. Otherwise, a run-time stack overflow can occur. The C compiler detects and reports overflows.

Default: `200000`

SupportNonFinite

Specify whether to generate nonfinite data and operations.

Default: `true`

TargetLang

Specify the target language. Set to `'C'` to generate C code. Set to `C++` to generate C++ code. If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Default: `'C'`

TargetLangStandard

Specify a standard math library for the generated code. Options include `'C89/C90 (ANSI)'`, `'C99 (ISO)'`, and `'C++03 (ISO)'`.

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.

Default: `'C89/C90 (ANSI)'` for C, `'C++03 (ISO)'` for C++.

Toolchain

Specify the toolchain to use. If you do not specify a toolchain, MATLAB Coder automatically locates an installed toolchain.

Default: `'Automatically locate an installed toolchain'`

Verbose

Display code generation progress.

Default: `false`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Examples

Generate a standalone C/C++ static library from a MATLAB function that is suitable for code generation:

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 static library.

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

3 Generate the C library files in the default folder (codegen/lib/coderand). Use the -config option to specify the configuration object.

```
codegen -config cfg coderand
```

Generate a C executable file from a MATLAB function that is suitable for code generation. Specify the main C function as a configuration parameter.

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

```
function r = coderand() %#codegen
```

```
r = rand();
```

2 Write a main C function, c:\myfiles\main.c, that calls coderand.

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"

int main()
{
    coderand_initialize();

    printf("coderand=%g\n", coderand());

    coderand_terminate();

    return 0;
}
```

3 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 coderand
```

codegen generates C executables and supporting files in the default folder codegen/exe/coderand.

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 on the command line. You can use a source, object, or library file.

Alternatives

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

See Also

codegen | coder | coder.EmbeddedCodeConfig

Topics

coder.Constant class

Package: coder
Superclasses: coder.Type

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) in the MATLAB documentation.

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 `identity` that copies its input to its output.

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

Create a code configuration object for MEX code generation.

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

Generate a MEX function `identity_mex` with the constant input 42.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex`. You must provide the input 42.

```
identity_mex(42)

ans =

    42
```

Configure `ConstantInputs` so that the MEX function does not check that the input value matches the compile-time value.

```
cfg.ConstantInputs = 'IgnoreValues';
```

Generate `identity_mex` with the new configuration.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex` with a constant input value other than `42` .

```
identity_mex(50)
```

```
ans =

    42
```

The MEX function ignored the input value `50`.

Configure `ConstantInputs` so that the MEX function does not include the constant input.

```
cfg.ConstantInputs = 'Remove';
```

Generate `identity_mex` with the new configuration.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex`. Do not provide the input value .

```
identity_mex()
```

```
ans =

    42
```

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

- `coder.Constant` does not support value class objects.

See Also

See Also
codegen | coder.newtype | coder.Type

Topics
"Specify Constant Inputs at the Command Line"
"Control Constant Inputs in MEX Function Signatures"
"Define Constant Global Data"

Introduced in R2011a

coder.EmbeddedCodeConfig class

Package: coder
Superclasses: coder.CodeConfig

`codegen` configuration object that specifies code generation parameters for code generation with an Embedded Coder license

Description

A `coder.EmbeddedCodeConfig` object contains the configuration parameters that the `codegen` function requires to generate standalone C/C++ libraries and executables for an embedded target. Use the `-config` option to pass this object to the `codegen` function.

Construction

cfg = `coder.config('lib')` creates a code generation configuration object for C/C++ static library generation. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` object.

cfg = `coder.config('dll')` creates a code generation configuration object for C/C++ dynamic library generation. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` object.

cfg = `coder.config('exe')` creates a code generation configuration object for C/C++ executable generation. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object. Otherwise, it creates a `coder.CodeConfig` object.

cfg = `coder.config(`*output_type*`, 'ecoder', false)` creates a `coder.CodeConfig` object for the specified output type even if the Embedded Coder product is installed.

cfg = `coder.config(`*output_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 an Embedded Coder license is available.

Properties

BuildConfiguration

Specify build configuration. 'Faster Builds', 'Faster Runs', 'Debug', 'Specify'.

Default: 'Faster Builds'

CastingMode

Specify data type casting level for variables in the generated C/C++ code.

Value	Description

'Nominal'	Generate 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'	Generate 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;
	}
	``

'Explicit'	Generate C/C++ code that casts data type values explicitly. For example:

Value	Description
| | ```c
short addone(short x)
{
  int i0;
  i0 = (int)x + 1;
  if (i0 > 32767) {
    i0 = 32767;
  }

  return (short)i0;
}
``` |

Default: `'Nominal'`

CodeExecutionProfiling

Enable execution-time profiling during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution.

Default: `false`

CodeTemplate

Specify a code generation template for file and function banners in the generated code. This parameter is a handle to a `coder.MATLABCodeTemplate` object constructed from a code generation template (CGT) file.

This parameter is empty by default. If you do not set this parameter to a `coder.MATLABCodeTemplate` object, the code generator produces default banners.

CodeReplacementLibrary

Specify a code replacement library for the generated code.

Value	Description
`'None'`	Does not use a code replacement library.
`'GNU C99 extensions'`	Generates calls to the GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
`'Intel IPP for x86-64 (Windows)'`	Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.

Value	Description
`'Intel IPP/SSE for x86-64 (Windows)'`	Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
`'Intel IPP for x86-64 (Windows using MinGW compiler)'`	Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.
`'Intel IPP/SSE for x86-64 (Windows using MinGW compiler)'`	Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
`'Intel IPP for x86/Pentium (Windows)'`	Generates calls to the IPP library for the x86/Pentium Windows platform.
`'Intel IPP/SSE x86/Pentium (Windows)'`	Generates calls to the IPP and SSE libraries for the x86/Pentium Windows platform.
`'Intel IPP for x86-64 (Linux)'`	Generates calls to the IPP library for the x86-64 Linux platform.
`'Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)'`	Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

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.

Default: `'None'`

CommentStyle

Specify comment style in the generated C or C++ code.

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.

Default: `'Auto'`

CompileTimeRecursionLimit

For compile-time recursion, control the number of copies of a function that are allowed in the generated code. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to 0. The default compile-time recursion limit is high 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".

Default: *integer*, 50

ConvertIfToSwitch

Select whether to convert `if-elseif-else` patterns to `switch-case` statements. This optimization works only for integer and enumerated type inputs.

Default: false

ConstantFoldingTimeout

Specify the maximum number of instructions that the constant folder executes before stopping. In some situations, code generation requires specific instructions to be constant. If code generation is failing, increase this value.

Default: 10000

CustomHeaderCode

Specify code to appear near the top of each C/C++ header file generated from your MATLAB algorithm code. Specify code as a character vector.

Default: ''

CustomInclude

Specify a space-separated list of include folders to add to the include path when compiling the generated code. Specify list of include folders as a character vector.

If your list includes Windows paths that contain spaces, enclose each instance in double quotes, for example:

'C:\Project "C:\Custom Files"'

Default: ''

CustomInitializer

Specify code to appear in the initialize function of the generated .c or .cpp file. Specify code as a character vector.

Default: ''

CustomLAPACKCallback

Specify the name of a LAPACK callback class that derives from coder.LAPACKCallback. Specify name 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.

Default: `' '`

CustomLibrary

Specify a space-separated list of static library files to link with the generated code. Specify list as a character vector.

Default: `' '`

CustomSource

Specify a space-separated list of source files to compile and link with the generated code. Specify the list of source files as a character vector.

The build process looks 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 intended. For example, suppose that you specify `foo.cpp` as a source file. If `foo.c` and `foo.cpp` are in folders on the search path, you cannot be sure whether the build process uses `foo.c` or `foo.cpp`.

Default: `' '`

CustomSourceCode

Specify code to appear near the top of the generated `.c` or `.cpp` file, outside of a function. Specify list as a character vector.

Do not specify a C static function definition.

Default: `' '`

CustomSymbolStrEMXArray

Customize generated identifiers for EMX Array types (Embeddable mxArray types). See "Settings" on page 3-69.

Default: `'emxArray_$M$N'`

CustomSymbolStrEMXArrayFcn

Customize generated identifiers for EMX Array (Embeddable mxArrays) utility functions. See "Settings" on page 3-69.

Default: `'emx$M$N'`

CustomSymbolStrFcn

Customize generated local function identifiers. See "Settings" on page 3-69.

Default: `'m_$M$N'`

CustomSymbolStrField

Customize generated field names in global type identifiers. See "Settings" on page 3-69.

Default: `'$M$N'`

CustomSymbolStrGlobalVar

Customize generated global variable identifiers. See "Settings" on page 3-69.

Default: `'$M$N'`

CustomSymbolStrMacro

Customize generated constant macro identifiers. See "Settings" on page 3-69.

Default: `'$M$N'`

CustomSymbolStrTmpVar

Customize generated local temporary variable identifiers. See "Settings" on page 3-69.

Default: `'$M$N'`

CustomSymbolStrType

Customize generated global type identifiers. See "Settings" on page 3-69.

Default: `'$M$N'`

CustomTerminator

Specify code to appear in the terminate function of the generated `.c` or `.cpp` file. Specify code as a character vector.

Default: `''`

CustomToolchainOptions

Specify custom settings for each tool in the selected toolchain, as a cell array.

Dependencies:

- `Toolchain` determines which tools and options appear in the cell.
- Setting `BuildConfiguration` to `Specify` enables the options specified by `CustomToolchainOptions`.

Start by getting the current options and values. For example:

```
rtwdemo_sil_topmodel;
set_param(gcs, 'BuildConfiguration', 'Specify')
opt = get_param(gcs, '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".

Default: *cell array*

DataTypeReplacement

Specify whether to use built-in C data types or predefined types from `rtwtypes.h` in generated code.

Set to `'CoderTypeDefs'` to use the data types from `rtwtypes.h`. Otherwise, to use built-in C data types, retain default value or set to `'CBuiltIn'`.

Default: `'CBuiltIn'`

Description

Description of the `coder.EmbeddedCodeConfig` object, specified as a character vector.

Default: `'class EmbeddedCodeConfig: C code generation Ecoder configuration objects'`

DynamicMemoryAllocation

Control use of dynamic memory allocation for variable-size data.

By default, dynamic memory allocation is enabled for variable-size arrays whose size (in bytes) is greater than or equal to DynamicMemoryAllocationThreshold. codegen allocates memory for this variable-size data dynamically on the heap.

Set this property to `'Off'` to allocate memory statically on the stack. Set it to `'AllVariableSizeArrays'` to allocate memory for all variable-size arrays dynamically on the heap. You **must** use dynamic memory allocation for unbounded, variable-size data.

Dependencies:

- EnableVariableSizing enables this parameter.
- Setting this parameter to `'Threshold'` enables the DynamicMemoryAllocationThreshold parameter.

Default: `'Threshold'`

DynamicMemoryAllocationThreshold

Specify the size threshold in bytes. codegen allocates memory on the heap for variable-size arrays whose size is greater than or equal to this threshold.

Dependency:

- Setting DynamicMemoryAllocation to `'Threshold'` enables this parameter.

Default: 65536

EnableAutoExtrinsicCalls

Specify whether MATLAB Coder must treat common visualization functions as extrinsic functions. When this option is enabled, MATLAB Coder detects calls to many common visualization functions, such as plot, disp, and figure. For MEX code generation, MATLAB Coder calls out to MATLAB for these functions. For standalone code generation, MATLAB Coder does not generate code for these visualization functions. This capability reduces the amount of time that you spend making your code suitable for code generation. It also removes the requirement to declare these functions extrinsic using the coder.extrinsic function.

Default: true

EnableMemcpy

Enable the `memcpy` optimization. To optimize code that copies consecutive array elements, the `memcpy` optimization replaces the code with a `memcpy` call. A `memcpy` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments. The code generator invokes the `memcpy` optimization when the following conditions are true:

- `EnableMemcpy` is `true`.
- The number of bytes to copy is greater than or equal to `MemcpyThreshold`. The number of bytes to copy is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

See "memcpy Optimization".

Default: `true`

EnableOpenMP

If possible, enable OpenMP. Using the OpenMP library, the C/C++ code that MATLAB Coder generates for `parfor`-loops can run on multiple threads. With OpenMP disabled, MATLAB Coder treats `parfor`-loops as for-loops and generates C/C++ code that runs on a single thread.

Default: `true`

EnableRuntimeRecursion

Allow recursive functions in generated code. If your MATLAB code requires run-time recursion and this parameter is `false`, code generation fails.

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`.

Default: `true`

EnableSignedLeftShifts

Specify whether to replace multiplications by powers of two with signed left bitwise shifts in the generated C/C++ code. 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 this option to `false`.

When this option is `true`, MATLAB Coder uses signed left shifts for multiplication by powers of two. Here is an example of generated C code that uses signed left shift for multiplication by eight:

```
i <<= 3;
```

When this option is `false`, MATLAB Coder does not use signed left shifts for multiplication by powers of two. Here is an example of generated C code that does not use signed left shift for multiplication by eight:

```
i = i * 8;
```

Default: `true`

EnableSignedRightShifts

Specify whether to allow signed right bitwise shifts in the generated C/C++ code. 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 this option to `false`.

When this option is `true`, MATLAB Coder uses signed right shifts. Here is an example of generated C code that uses a signed right shift:

```
i >>= 3
```

When this option is `false`, MATLAB Coder replaces right shifts on signed integers with a function call in the generated code.

```
i = asr_s32(i, 3U);
```

Default: `true`

EnableStrengthReduction

Enable optimization that simplifies array indexing in loops in the generated code. When possible, for array indices in loops, this optimization replaces multiply operations with add operations. Multiply operations can be expensive. This optimization is useful when the C/C++ compiler on the target platform does not optimize the array indexing. Even

when the optimization replaces the multiply operations in the generated code, it is possible that the C/C++ compiler can generate multiply instructions.

Default: `false`

EnableVariableSizing

Enable support for variable-size arrays.

Dependency:

• This parameter enables the parameter `DynamicMemoryAllocation`.

Default: `true`

FilePartitionMethod

Specify whether to generate separate C/C++ files for each MATLAB language file (`'MapMFileToCFile'`) or generate a single file for C/C++ functions that map to your MATLAB entry-point functions (`'SingleFile'`). When you choose `'SingleFile'`, the code generator creates separate C/C++ files for utility functions.

Default: `'MapMFileToCFile'`

GenerateCodeMetricsReport

Generate a static code metrics report including generated file information, number of lines, and memory usage.

Default: `false`

GenCodeOnly

Specify code generation versus an executable or library build.

Default: `false`

GenerateCodeReplacementReport

Generate a code replacements report that summarizes the replacements used 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.

Default: `false`

GenerateComments

Place comments in the generated files.

Dependencies:

- Enables `CommentStyle`.
- Enables `MATLABFcnDesc`.
- Enables `MATLABSourceComments`.

Default: `true`

GenerateExampleMain

Specify whether to generate an example C/C++ main function. If example main generation is enabled, MATLAB Coder generates source and header files for the main function in the `examples` subfolder of the build folder. For C code generation, it generates the files `main.c` and `main.h`. For C++ code generation, it generates the files `main.cpp` and `main.h`.

The example main function declares and initializes data. It calls entry-point functions but does not use values returned from the entry-point functions.

Before you use the example main files, copy them to another location and modify them to meet the requirements of your application.

Value	Description
`'DoNotGenerate'`	Does not generate an example C/C++ main function.
`'GenerateCodeOnly'`	Generates an example C/C++ main function but does not compile it.
`'GenerateCodeAndCompile'`	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`, MATLAB Coder does not compile the C/C++ main function.

Default: `'GenerateCodeOnly'`

GenerateMakefile

Specify whether to generate a makefile during the build process.

Default: `true`

GenerateReport

Default: `false`

Hardware

Handle to `coder.Hardware` object that specifies the hardware board for processor-in-the-loop (PIL) execution. Use `coder.hardware` to create the `coder.Hardware` object. For example:

```
cfg = coder.config('lib','ecoder',true);
hw = coder.hardware('BeagleBone Black');
cfg.Hardware = hw;
```

Dependencies:

• Setting `Hardware` sets `HardwareImplementation` to a
  `coder.HardwareImplementation` object customized for the hardware specified by
  `Hardware`.

HardwareImplementation

Handle to `coder.HardwareImplementation` object that specifies hardware-specific configuration parameters for C/C++ code generation.

If you set the `Hardware` property, `HardwareImplementation` is set to a `coder.HardwareImplementation` object customized for the hardware that is specified by `Hardware`. If you do not set the `Hardware` property, `HardwareImplementation` defaults to a `coder.HardwareImplementation` object with properties that are set for the MATLAB host computer.

Dependency:

Setting `Hardware` sets `HardwareImplementation` to a `coder.HardwareImplementation` object customized for the hardware that is specified by `Hardware`.

`HighlightPotentialDataTypeIssues`

Highlight potential data type issues in the code generation report. If this option is enabled, 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 a Fixed-Point Designer™ license, the report also highlights expressions in the MATLAB code that result in expensive fixed-point operations in the generated code.

Default: `false`

`IncludeTerminateFcn`

Generate a terminate function.

If you set this property to `false` but a terminate function is required, for example, to free memory, MATLAB Coder issues a warning.

Default: `true`

`IndentSize`

Specify the number of characters per indentation level. Specify an integer from 2 to 8.

Default: 2

`IndentStyle`

Specify the style for the placement of braces in the generated C/C++ code.

Value	Description

`'K&R'`	For blocks within a function, an opening brace is on the same line as its control statement. For example:
	
	``c
	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;`
	`  }`
	`}`

Default: `'K&R'`

InitFltsAndDblsToZero

Enable the `memset` optimization for assignment of float or double 0 to consecutive array elements. To optimize code that assigns a constant value to consecutive array elements, the `memset` optimization replaces the code with a `memset` call. A `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments. To assign float or double 0 to consecutive array elements, the code generator invokes the `memset` optimization when the following conditions are true:

- `InitFltsAndDblsToZero` is `true`.
- The number of bytes to assign is greater than or equal to `MemcpyThreshold`. The number of bytes to assign is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

See "memset Optimization".

Default: `true`

InlineStackLimit

Specify the stack size limit on inlined functions. This specification determines the amount of stack space allocated for local variables of the inlined function.

Specifying a limit on 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 feature is especially important for embedded processors, where stack size can be limited.

Default: 4000

InlineThreshold

Specify function size for inline threshold. Unless there are conflicts with other inlining conditions, MATLAB Coder inlines functions that are smaller than this size.

The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. You must experiment with this parameter to obtain the inlining behavior that you want. For instance, if the default setting for this parameter is leading to large functions being inlined, you can tune the parameter in steps until you are satisfied with the inlining behavior.

Default: 10

InlineThresholdMax

Specify the maximum size of functions after inlining. If the size of the calling function after inlining exceeds `InlineThresholdMax`, MATLAB Coder does not inline the called function.

The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. You must experiment with this parameter to obtain the inlining behavior that you want. For instance, if the default setting for this parameter is leading to the inlining of large functions, you can tune the parameter in steps until you are satisfied with the inlining behavior.

Default: 200

LaunchReport

Specify whether to display a report after code generation is complete or an error occurs.

Default: true

MATLABFcnDesc

Include MATLAB function help text in a function banner in generated code. If not selected, MATLAB Coder treats the help text as a user comment.

Dependencies:

- `GenerateComments` enables this parameter.

Default: `true`

MATLABSourceComments

Include MATLAB source code as comments in the generated code.

Dependencies:

- `GenerateComments` enables this parameter.

Default: `false`

MaxIdLength

Specify maximum number of characters in 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 parameter 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.

Minimum is 31. Maximum is 256.

Default: 31

MemcpyThreshold

Specify the minimum number of bytes for the code generator to invoke the `memcpy` optimization or the `memset` optimization. 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. A `memcpy` or `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

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.

See "memcpy Optimization" and "memset Optimization".

Default: 64

MultiInstanceCode

Generate reusable, multi-instance code that is reentrant.

Default: false

Name

Name of the configuration object.

Default: *character vector*, 'EmbeddedCodeConfig'

OutputType

Specify whether to generate a standalone C/C++ static library, dynamic library, or executable. Set to 'LIB' to generate a static library, 'DLL' to generate a dynamic library or 'EXE' to generate an executable.

Default: 'LIB'

ParenthesesLevel

Specify the parenthesization level in the generated C/C++ code.

Value	Description
'Minimum'	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 using the minimum level, for certain settings in some compilers, you can receive compiler warnings. To eliminate these warnings, try the nominal level.
'Nominal'	Inserts parentheses to balance readability and visual complexity. For example: `Out = ((In2 - In1 > 1.0) && (In2 > 2.0));`

Value	Description
`'Maximum'`	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));`

Default: `'Nominal'`

`PassStructByReference`

Specify whether to pass structures by reference to entry-point functions. Set to `true` to pass structures by reference. You thereby reduce memory usage and execution time by minimizing the number of copies of parameters at entry-point function boundaries. Set to `false` to pass structures by value.

This parameter applies only to entry-point functions.

If you set this parameter to `true`, an entry-point function that writes to a field of a structure parameter overwrites the input value.

Default: `true`

`PostCodeGenCommand`

Specify command to customize build processing after code generation using `codegen`. Specify command as a character vector.

Default: `''`

`PreserveExternInFcnDecls`

Specify whether the declarations of external functions generated by `codegen` include the `extern` keyword.

Default: `true`

`PreserveVariableNames`

Specify which variable names the code generator must preserve in the generated code.

Value	Description
`'None'`	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.

Default: `'None'`

PurelyIntegerCode

Specify whether to generate floating-point data and operations.

Default: false

ReportPotentialDifferences

Specify whether to report potential behavior differences between generated code and MATLAB code. If `ReportPotentialDifferences` is `true`, the code generation report has a tab that lists the potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

Default: `true`

ReservedNameArray

Enter a space-separated list of names that MATLAB Coder is not to use for naming functions or variables. Specify list as a character vector.

Default: `''`

RuntimeChecks

Enable run-time error detection and reporting in the generated C/C++ code. If you select this option, 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.

Default: `false`

SaturateOnIntegerOverflow

Overflows saturate to either the minimum or maximum value that the data type can represent. Otherwise, the overflow behavior depends on your target C compiler. 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.

Default: `true`

SILDebugging

Enable debugger to observe code behavior during a software-in-the-loop (SIL) execution.

The software supports the following debuggers:

- On Windows, Microsoft Visual C++® debugger.
- On Linux, GNU Data Display Debugger (DDD).

Default: `false`

StackUsageMax

Specify the maximum stack usage per application in bytes. Set a limit that is lower than the available stack size. Otherwise, a run-time stack overflow can occur. The C compiler detects and reports overflows.

Default: `200000`

SupportNonFinite

Specify whether to generate nonfinite data and operations.

Default: `true`

TargetLang

Specify the target language. Set to `'C'` to generate C code. Set to `C++` to generate C++ code. If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Default: `'C'`

TargetLangStandard

Specify a standard math library for the generated code. Options include `'C89/C90 (ANSI)'`, `'C99 (ISO)'`, and `'C++03 (ISO)'`.

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.

Default: `'C89/C90 (ANSI)'` for C, `'C++03 (ISO)'` for C++.

Toolchain

Specify the toolchain to use. If you do not specify a toolchain, MATLAB Coder automatically locates an installed toolchain.

Default: `'Automatically locate an installed toolchain'`

Verbose

Display code generation progress.

Default: `false`

VerificationMode

Specify code verification mode.

- `'None'` — Normal execution
- `'SIL'` — Software-in-the-loop (SIL) execution
- `'PIL'` — Processor-in-the-loop (PIL) execution

Default: `'None'`

Settings

Enter a macro that specifies the format of the generated identifier. The macro can include valid C identifier characters and a combination of the following format tokens:

Token	Description
$M	Code generator inserts name mangling text to avoid naming collisions. Required.
$N	Code generator inserts name of object (global variable, global type, local function, local temporary variable, or constant macro) for which identifier is generated. Improves readability of generated code.
$R	Code generator inserts root project name into identifier, replacing unsupported characters with the underscore (_) character.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Examples

Generate a standalone C/C++ static library from a MATLAB function that is suitable for code generation.

Note: To generate code for this example, you must have an Embedded Coder license.

---

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 static library.

```
cfg = coder.config('lib')
```
This command creates a `coder.EmbeddedCodeConfig` object.

3 Set the `PurelyIntegerCode` parameter to `true` to enable generation of integer-only code.

```
cfg.PurelyIntegerCode = true;
```

4 Generate the C library files in the default folder (codegen/lib/coderand). Use the -`config` option to specify the configuration object.

```
codegen -config cfg coderand
```

Alternatives

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

See Also

codegen | coder | coder.config

Topics

coder.EnumType class

Package: coder
Superclasses: coder.ArrayType

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 a numeration 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) in the MATLAB documentation.

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

coder.ArrayType | `coder.newtype` | coder.Type | `coder.typeof` | `coder.resize` | `codegen`

Topics
"Enumerations"

Introduced in R2011a

coder.ExternalDependency class

Package: coder

Interface to external code

Description

`coder.ExternalDependency` is an abstract class for encapsulating the interface between external code and MATLAB code intended for code generation. You 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. The derived class contains information about external file locations, build information, and the programming interface to external functions.

To define a class, `myclass`, make the following line the first line of your class definition file:

```
classdef myclass < coder.ExternalDependency
```

You must define all of the methods listed in "Methods" on page 3-76. These methods are static and are not compiled. When you write these methods, use `coder.BuildConfig` methods to access build information.

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. Suppose you define the following method for a class named `AdderAPI`:

```
function c = adder(a, b)
    coder.cinclude('adder.h');
    c = 0;
    c = coder.ceval('adder', a, b);
end
```

This method defines the interface to a function `adder` which has two inputs `a` and `b`. In your MATLAB code, call `adder` this way:

```
y = AdderAPI.adder(x1, x2);
```

Methods

Examples

Encapsulate the interface to an external C dynamic linked 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(ctx)
            if  ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
```

```matlab
        end

        function updateBuildInfo(buildInfo, ctx)
            [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

            % Header files
            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
            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
                % running in generated code, call library function
                coder.cinclude('adder.h');

                % Because MATLAB Coder generated adder, use the
                % housekeeping functions before and after calling
                % adder with coder.ceval.
                % Call initialize function before calling adder for the
                % first time.

                coder.ceval('adder_initialize');
                c = 0;
                c = coder.ceval('adder', a, b);
```

```matlab
                % Call the terminate function after
                % calling adder for the last time.

                coder.ceval('adder_terminate');
            end
        end
    end
end
```

Write a function `adder_main` that calls the external library function `adder`.

```matlab
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.

```matlab
codegen('adder_main', '-args', {7,9}, '-report')
```

Copy the library to the current folder using the file extension for your platform.

For Windows, use:

```matlab
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dll'));
```

For Linux, use:

```matlab
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.so'));
```

Run the MEX function and verify the result.

```matlab
adder_main_mex(2,3)
```

See Also

See Also
coder.BuildConfig | coder.ceval | coder.cinclude | coder.updateBuildInfo

Topics
"Encapsulating the Interface to External Code"

"Best Practices for Using coder.ExternalDependency"
"Build Information Object"
"Build Information Methods"

Introduced in R2013b

coder.FiType class

Package: coder
Superclasses: coder.ArrayType

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. `Name` must appear inside single quotes (`' '`). 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 single 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) in the MATLAB documentation.

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

coder.ArrayType | `coder.resize` | coder.Type | `coder.typeof` | `coder.newtype` | `codegen`

Topics

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

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
```

- "C Code Generation at the Command Line"

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

See Also
coder.codeConfig | codegen | coder | coder.config

Topics
"C Code Generation at the Command Line"

coder.HardwareImplementation class

Package: coder

`codegen` configuration object that specifies hardware implementation parameters for code generation

Description

A `coder.HardwareImplementation` object contains hardware-specific configuration parameters. The `codegen` function uses these parameters to generate standalone C/C ++ libraries and executables for specific target hardware. To use this object, refer to it from the related coder.CodeConfig or coder.EmbeddedCodeConfig object that `codegen` is using.

Construction

`hw_cfg = coder.HardwareImplementation` creates a `coder.HardwareImplementation` object.

Properties

Description

Description of hardware implementation object.

character vector, '**class HardwareImplementation: Hardware implementation specifications.**', Maximum Length: 78 characters

Name

Name of hardware implementation object.

character vector, '**HardwareImplementation**', Maximum Length: 22 characters

ProdBitPerChar

Describe length in bits of the C `char` data type that the deployment hardware supports.

Value must be a multiple of 8 between 8 and 32.

Dependencies:

- Specifying a device using the `ProdHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **8**

ProdBitPerDouble

Describe the bit length of C `double` data type that the deployment hardware supports (read only).

double, **64**

ProdBitPerFloat

Describe the bit length of C floating-point data type that the deployment hardware supports (read only).

double, **32**

ProdBitPerInt

Describe length in bits of the C `int` data type that the deployment hardware supports.

Value must be a multiple of 8 between 8 and 32.

Dependencies:

- Specifying a device using the `ProdHWDeviceType` parameter sets a device-specific value for this parameter.

• This parameter is enabled only if you can modify it for the specified device.

integer, **32**

ProdBitPerLong

Describe length in bits of the C `long` data type that the deployment hardware supports.

Value must be a multiple of 8 between 32 and 128.

Dependencies:

• Specifying a device using the `ProdHWDeviceType` parameter sets a device-specific value for this parameter.
• This parameter is enabled only if you can modify it for the specified device.

integer, **32**

ProdBitPerLongLong

Describe length in bits of the C `long long` data type that the deployment hardware supports.

Tips:

• Use the C `long long` data type only if your C compiler supports `long long`.
• You can change the value of this parameter only for custom targets. For custom targets, values must be a multiple of 8 and between 64 and 128.

Dependencies:

• `ProdLongLongMode` enables use of this parameter.
• The value of this parameter must be greater than or equal to the value of `ProdBitPerLong`.
• Selecting a device using the `ProdHWDeviceType` parameter sets a device-specific value for this parameter.
• This parameter is enabled only if you can modify it for the specified device.

integer, **64**

ProdBitPerPointer

Describe the bit-length of pointer data for the deployment hardware (read only).

integer, **64**

ProdBitPerSizeT

Describe the bit-length of `size_t` data for the deployment hardware.

If `ProdEqTarget` is `true`, Embedded Coder processor-in-the-loop (PIL) execution checks this setting with reference to the target hardware. If `ProdEqTarget` is `false`, the PIL execution checks the `TargetBitPerSizeT` setting with reference to the target hardware..

Value must be 8, 16, 24, 32, 40, 64, or 128 *and* greater or equal to the value of int.

Dependencies:

- Selecting a device using the **Device vendor** and **Device type** parameters sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

ProdBitPerPtrDiffT

Describe the bit-length of `ptrdiff_t` data for the deployment hardware.

If `ProdEqTarget` is `true`, Embedded Coder processor-in-the-loop (PIL) execution checks this setting with reference to the target hardware. If `ProdEqTarget` is `false`, the PIL execution checks the `TargetBitPerPtrDiffT` setting with reference to the target hardware.

Value must be 8, 16, 24, 32, 40, 64, or 128 *and* greater or equal to the value of int.

Dependencies:

- Selecting a device using the **Device vendor** and **Device type** parameters sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

ProdBitPerShort

Describe length in bits of the C `short` data type that the deployment hardware supports.

Value must be a multiple of 8 between 8 and 32.

Dependencies:

- Selecting a device using the `ProdHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **16**

ProdEndianess

Describe significance of the first byte of a data word for the deployment hardware.

character vector, `'Unspecified'`, **`'LittleEndian'`**, `'BigEndian'`

ProdEqTarget

Specify whether the test hardware differs from the deployment hardware.

Dependencies:

- Setting this parameter to `true` disables the target properties.
- Setting this parameter to `false` enables the target properties that specify the test hardware properties.

true, false

ProdHWDeviceType:

Specify manufacturer and type of hardware that you use to implement the production version of the system.

Because `codegen` cannot generate code for ASICs or FPGAs, if `ProdHWDeviceType` is set to ASIC/FPGA, `TestHWDeviceType` is automatically set to `'Generic->MATLAB Host Computer'`.

character vector, '**Generic->MATLAB Host Computer**'

ProdIntDivRoundTo

Describe how your compiler rounds the result of dividing one signed integer by another to produce a signed integer quotient.

character vector, 'Undefined', '**Zero**', 'Floor'

ProdLargestAtomicFloat

Specify the largest floating-point data type that can be atomically loaded and stored on the deployment hardware.

Dependencies:

- Specifying a device using the ProdHWDeviceType parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

character vector, '**None**'

ProdLargestAtomicInteger

Specify the largest integer data type that can be atomically loaded and stored on the deployment hardware.

Dependencies:

- Specifying a device using the ProdHWDeviceType parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.
- You can set this parameter to long long only if the deployment hardware supports the C long long data type and you have set the ProdLongLongMode parameter to true.

character vector, '**Char**'

ProdLongLongMode

Specify that your C compiler supports the C long long data type. Most C99 compilers support long long. Set to true to enable use of the C long long data type for code

generation for the deployment hardware. Set to `false` to disable the use of C `long long` data type for code generation for the deployment hardware.

Tips:

- This parameter is enabled only if the specified deployment hardware supports the C `long long` data type.
- If your compiler does not support C `long long`, do not select this parameter.

Dependency:

- This parameter enables use of `ProdBitPerLongLong`.

true, `false`

ProdShiftRightIntArith

Describe whether your compiler implements a signed integer right shift as an arithmetic right shift.

true, `false`

ProdWordSize

Describe microprocessor native word size for the deployment hardware.

Value must be a multiple of 8 between 8 and 64.

Dependencies:

- Selecting a device using the `ProdHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **64**

TargetBitPerChar

Describe length in bits of the C `char` data type that the test hardware supports.

Value must be a multiple of 8 between 8 and 32.

Dependencies:

- Specifying a device using the `TargetHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **8**

TargetBitPerDouble

Describe the bit length of C `double` data type that the test hardware supports (read only).

integer, **64**

TargetBitPerFloat

Describe the bit length of C floating-point data type that the test hardware supports (read only).

integer, **32**

TargetBitPerInt

Describe length in bits of the C `int` data type that the test hardware supports.

Value must be a multiple of 8 between 8 and 32.

Dependencies:

- Specifying a device using the `TargetHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **32**

TargetBitPerLong

Describe length in bits of the C `long` data type that the test hardware supports.

Value must be a multiple of 8 between 32 and 128.

Dependencies:

- Specifying a device using the `TargetHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **32**

TargetBitPerLongLong

Describe length in bits of the C `long long` data type that the test hardware supports.

Dependencies:

- `TargetLongLongMode` enables use of this parameter.
- The value of this parameter must be greater than or equal to the value of `TargetBitPerLong`.

Tips:

- Use the C `long long` data type only if your C compiler supports `long long`.
- Change the value of this parameter for custom targets only. For custom targets, values must be a multiple of 8 and between 64 and 128.

integer, **64**

TargetBitPerPointer

Describe the bit-length of pointer data for the test hardware (read only).

integer, **64**

TargetBitPerSizeT

Describe the bit-length of `size_t` data for the test hardware.

If `ProdEqTarget` is `false`, Embedded Coder processor-in-the-loop (PIL) execution checks this setting with reference to the target hardware. If `ProdEqTarget` is `true`, the PIL execution checks the `ProdBitPerSizeT` setting with reference to the target hardware..

Value must be 8, 16, 24, 32, 40, 64, or 128 *and* greater or equal to the value of int.

Dependencies:

- Selecting a device using the **Device vendor** and **Device type** parameters sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

TargetBitPerPtrDiffT

Describe the bit-length of `ptrdiff_t` data for the test hardware.

If `ProdEqTarget` is `false`, Embedded Coder processor-in-the-loop (PIL) execution checks this setting with reference to the target hardware. If `ProdEqTarget` is `true`, the PIL execution checks the `ProdBitPerPtrDiffT` setting with reference to the target hardware.

Value must be 8, 16, 24, 32, 40, 64, or 128 *and* greater or equal to the value of int.

Dependencies:

- Selecting a device using the **Device vendor** and **Device type** parameters sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

TargetBitPerShort

Describe length in bits of the C `short` data type that the test hardware supports.

Value must be a multiple of 8 between 8 and 32 .

Dependencies:

- Selecting a device using the `TargetHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **16**

TargetEndianess

Describe significance of the first byte of a data word for the test hardware.

character vector, 'Unspecified', **'LittleEndian'**, 'BigEndian'

TargetHWDeviceType:

Specify manufacturer and type of the hardware that you use to test the generated code.

Because `codegen` cannot generate code for ASICs or FPGAs, if `ProdHWDeviceType` is set to `ASIC/FPGA`, `TestHWDeviceType` is automatically set to `'Generic->MATLAB Host Computer'`.

character vector, **'Generic->MATLAB Host Computer'**

TargetIntDivRoundTo

Describe how your compiler rounds the result of two signed integers for the test hardware.

character vector, 'Undefined', **'Zero'**, 'Floor'

TargetLargestAtomicFloat

Specify the largest floating-point data type that can be atomically loaded and stored on the test hardware.

Dependencies:

- Specifying a device using the `TargetHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

character vector, **'None'**

TargetLargestAtomicInteger

Specify the largest integer data type that can be atomically loaded and stored on the test hardware.

Dependencies:

- Specifying a device using the `TargetHWDeviceType` parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

- You can set this parameter to long long only if the test hardware supports the C long long data type and you have set the TargetLongLongMode parameter to true.

character vector, '**Char**'

TargetLongLongMode

Specify that your C compiler supports the C long long data type. Most C99 compilers support long long. Set to true to enable use of the C long long data type for code generation for the test hardware. Set to false to disable use of the C long long data type for code generation for the test hardware.

Tips:

- This parameter is enabled only if the specified test hardware supports the C long long data type.
- If your compiler does not support C long long, do not select this parameter.

Dependency:

This parameter enables use of TargetBitPerLongLong.

true, false

TargetShiftRightIntArith

Describe whether your compiler implements a signed integer right shift as an arithmetic right shift.

true, false

TargetWordSize

Describe microprocessor native word size for the test hardware.

Value must be a multiple of 8 between 8 and 64.

Dependencies:

- Selecting a device using theTargetHWDeviceType parameter sets a device-specific value for this parameter.
- This parameter is enabled only if you can modify it for the specified device.

integer, **64**

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Examples

Create a hardware implementation configuration object. Use the object to generate a C static library.

1   Create a hardware implementation configuration object.

    hw_cfg = coder.HardwareImplementation;

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

    cfg = coder.config('lib');

3   Associate the hardware implementation object with the code generation configuration object.

    cfg.HardwareImplementation = hw_cfg;

4   Generate a C library for a MATLAB function foo that has no input parameters. Specify the configuration object using the -config option:

    codegen -config cfg foo

Alternatives

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

See Also
codegen | coder.CodeConfig | coder | coder.EmbeddedCodeConfig

Topics

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, specify the name of the LAPACK callback class.

- If you generate code by using the MATLAB Coder `codegen` command, set the code configuration object property `CustomLAPACKCallback` to the name of the callback class.
- If you generate code by using the MATLAB Coder app, set **Custom LAPACK library callback** to the name of the callback class.
- If the code is in the MATLAB Function block and you have Simulink Coder, in the Configuration Parameters dialog box, in the **Code Generation** category, on the **All Parameters** tab, set **Custom LAPACK library callback** to the name of the callback class.

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 3-106. These methods are static and are not compiled.

Methods

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.

```matlab
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.

```matlab
buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
```

```
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
```

- "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)

See Also

See Also
coder.ExternalDependency | coder.BuildConfig

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.MexCodeConfig class

Package: coder

codegen configuration object that specifies MEX function generation parameters

Description

A coder.MexCodeConfig object contains the configuration parameters required by the codegen function to generate MEX functions. Use the -config option to pass the object to the codegen function.

Construction

cfg=coder.config creates a coder.MexCodeConfig object for MEX function generation.

cfg=coder.config('mex') creates a coder.MexCodeConfig object for MEX function generation.

Properties

CompileTimeRecursionLimit

For compile-time recursion, control the number of copies of a function that are allowed in the generated code. To disallow recursion in the MATLAB code, set CompileTimeRecursionLimit to 0. The default compile-time recursion limit is high 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".

Default: *integer*, 50

ConstantFoldingTimeout

Specify, as a positive integer, the maximum number of instructions to be executed by the constant folder.

Default: 10000

ConstantInputs

Specify whether to include constant inputs in the MEX function signature.

By default, (`'CheckValues'`), the MEX function signature contains the constant inputs. The run-time values of the constant inputs must match their compile-time values. This option allows you to use the same test file to run the original MATLAB algorithm and the MEX function. Selecting this option slows down execution of the MEX function.

If you specify `'IgnoreValues'`, the MEX function signature contains the constant inputs. The run-time values of the constant inputs are ignored and do not need to match their compile-time values. This option allows you to use the same test file to run the original MATLAB algorithm and the MEX function.

If you specify `'Remove'`, the MEX function signature does not contain the constant inputs and does not match the MATLAB signature. This option is provided for backwards compatibility.

Default: `'CheckValues'`

CustomHeaderCode

Specify code to appear near the top of each C/C++ header file generated from your MATLAB algorithm code. Specify code as a character vector.

Default: `''`

CustomInclude

Specify a space-separated list of include folders to add to the include path when compiling the generated code. Specify list of include folders as a character vector.

If your list includes Windows paths that contain spaces, enclose each instance in double quotes, for example:

`'C:\Project "C:\Custom Files"'`

Default: `''`

CustomInitializer

Specify code to appear in the initialize function of the generated `.c` or `.cpp` file. Specify code as a character vector.

Default: ' '

CustomLibrary

Specify a space-separated list of static library files to link with the generated code. Specify list as a character vector.

Default: ' '

CustomSource

Specify a space-separated list of source files to compile and link with the generated code. Specify the list of source files as a character vector.

The build process looks 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 intended. For example, suppose that you specify `foo.cpp` as a source file. If `foo.c` and `foo.cpp` are in folders on the search path, you cannot be sure whether the build process uses `foo.c` or `foo.cpp`.

Default: ' '

CustomSourceCode

Specify code to appear near the top of the generated `.c` or `.cpp` file, outside of a function. Specify code as a character vector.

Do not specify a C static function definition.

Default: ' '

CustomTerminator

Specify code to appear in the terminate function of the generated `.c` or `.cpp` file. Specify code as a character vector.

Default: ' '

Description

Description of object, ,specified as a character vector.

Default: ' '

DynamicMemoryAllocation

Control use of dynamic memory allocation for variable-size data.

By default, dynamic memory allocation is enabled for variable-size arrays whose size (in bytes) is greater than or equal to DynamicMemoryAllocationThreshold and codegen allocates memory for this variable-size data dynamically on the heap.

Set this property to 'Off' to allocate memory statically on the stack. Set it to 'AllVariableSizeArrays' to allocate memory for all arrays dynamically on the heap . You **must** use dynamic memory allocation for unbounded, variable-size data.

Dependencies:

• EnableVariableSizing enables this parameter.
• Setting this parameter to 'Threshold' enables the DynamicMemoryAllocationThreshold parameter.

Default: 'Threshold'

DynamicMemoryAllocationThreshold

Specify the size threshold in bytes. codegen allocates memory on the heap for variable-size arrays whose size is greater than or equal to this threshold.

Dependency:

• Setting DynamicMemoryAllocation to 'Threshold' enables this parameter.

Default: 65536

EchoExpressions

Specify whether or not actions that do not terminate with a semicolon appear in the MATLAB Command Window.

Default: true

EnableAutoExtrinsicCalls

Specify whether MATLAB Coder treats common visualization functions as extrinsic functions. When this option is enabled, MATLAB Coder detects calls to many common visualization functions, such as plot, disp, and figure. For MEX code generation, MATLAB Coder calls out to MATLAB for these functions. For standalone code

3-111

generation, MATLAB Coder does not generate code for these visualization functions. This capability reduces the amount of time that you spend making your code suitable for code generation. It also removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

Default: `true`

EnableDebugging

Specify whether to use the debug option for the C compiler. If you enable debug mode, the C compiler does not optimize the code. The compilation is faster, but the execution is slower.

Default: `false`

EnableJIT

Specify whether to use just-in-time (JIT) compilation for generation of MEX functions. 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 MATLAB Coder 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`, MATLAB Coder uses JIT compilation and treats the `parfor`-loops as `for`-loops.

Default: `false`

EnableMemcpy

Enable the `memcpy` optimization. To optimize code that copies consecutive array elements, the `memcpy` optimization replaces the code with a `memcpy` call. A `memcpy` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments. The code generator invokes the `memcpy` optimization when the following conditions are true:

- `EnableMemcpy` is `true`.
- The number of bytes to copy is greater than or equal to `MemcpyThreshold`. The number of bytes to copy is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

See "memcpy Optimization".

Default: `true`

EnableOpenMP

If possible, enable OpenMP. Using the OpenMP library, the MEX function that MATLAB Coder generates for `parfor`-loops can run on multiple threads. With OpenMP disabled, MATLAB Coder treats `parfor`-loops as for-loops and generates a MEX function that runs on a single thread.

Use of the OpenMP library is not compatible with just-in-time (JIT) compilation. If `EnableJIT` and `EnableOpenMP` are `true` and your code uses `parfor`, MATLAB Coder uses JIT compilation and treats the `parfor`-loops as `for`-loops.

Default: `true`

EnableRuntimeRecursion

Allow recursive functions in generated code. If your MATLAB code requires run-time recursion and this parameter is `false`, code generation fails.

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`.

Default: `true`

EnableVariableSizing

Enable support for variable-size arrays.

Dependency:

• Enables `Dynamic memory allocation`.

Default: `true`

ExtrinsicCalls

Allow calls to extrinsic functions.

An extrinsic function is a function on the MATLAB path that MATLAB Coder dispatches to MATLAB software for execution. MATLAB Coder does not compile or generate code for extrinsic functions.

When enabled (`true`), generates code for the call to a MATLAB function, but does not generate the function's internal code.

When disabled (`false`), ignores the extrinsic function. Does not generate code for the call to the MATLAB function — as long as the extrinsic function does not affect the output of the MATLAB function. Otherwise, issues a compilation error.

`ExtrinsicCalls` affects how MEX functions built by MATLAB Coder generate random numbers when using the MATLAB `rand`, `randi`, and `randn` functions. If extrinsic calls are enabled, the generated MEX function uses the MATLAB global random number stream to generate random numbers. If extrinsic calls are not enabled, the MEX function built with MATLAB Coder uses a self-contained random number generator.

If you disable extrinsic calls, 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, enable extrinsic function calls and generate the MEX function again.

Default: `true`

FilePartitionMethod

Specify whether to generate separate C/C++ files for each MATLAB language file (`'MapMFileToCFile'`) or generate a single file for C/C++ functions that map to your MATLAB entry-point functions (`'SingleFile'`). When you choose `'SingleFile'`, the code generator creates separate C/C++ files for utility functions.

Default: `'MapMFileToCFile'`

GenCodeOnly

Control whether to compile the generated MEX function C/C++ code to produce a MEX function.

Default: `false`

GenerateComments

Place comments in the generated files.

Default: `true`

GenerateReport

Document generated code in a report.

Default: `false`

GlobalDataSyncMethod

Controls synchronization of MEX function global data with the MATLAB global workspace. For constant global data, controls verification of consistency between the MEX function constant global data and the MATLAB global workspace.

Value	Description for Global Data	Description for Constant Global Data
`'SyncAlways'` (default)	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 in conjunction 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. The MEX function ends with an error if the global data values in the MATLAB global workspace are inconsistent with the compile-time constant global values in the MEX function. Use the `coder.extrinsic -sync:off` option to turn off consistency checks after specific extrinsic calls.
`'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 in conjunction with the `coder.extrinsic -sync:on` option to turn on synchronization for these calls.	Verifies constant global data at MEX function entry only. The MEX function ends with an error if the global data values in the MATLAB global workspace are inconsistent with the compile-time constant global values in the MEX function. 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 globals. Otherwise, inconsistencies between MATLAB and the MEX function can occur.	Disables consistency checks.

Default: `'SyncAlways'`

`InitFltsAndDblsToZero`

Enable the `memset` optimization for assignment of float or double 0 to consecutive array elements. To optimize code that assigns a constant value to consecutive array elements, the `memset` optimization replaces the code with a `memset` call. A `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments. To assign float or double 0 to consecutive array elements, the code generator invokes the `memset` optimization when the following conditions are true:

- `InitFltsAndDblsToZero` is `true`.
- The number of bytes to assign is greater than or equal to `MemcpyThreshold`. The number of bytes to assign is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

See "memset Optimization".

Default: `true`

`InlineStackLimit`

Specify the stack size limit on inlined functions. This specification determines the amount of stack space allocated for local variables of the inlined function.

Specifying a limit on 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 feature is especially important for embedded processors, where stack size can be limited.

Default: 4000

`InlineThreshold`

Specify function size for inline threshold. Unless there are conflicts with other inlining conditions, MATLAB Coder inlines functions that are smaller than this size.

The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. You must experiment with this parameter to obtain the inlining behavior that you want. For instance, if the default setting for this parameter is leading to large functions being inlined and in turn

generating large C code, you can tune the parameter in steps until you are satisfied with the size of generated code.

Default: 10

InlineThresholdMax

Specify the maximum size of functions after inlining. If the size of the calling function after inlining exceeds `InlineThresholdMax`, MATLAB Coder does not inline the called function.

The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. You must experiment with this parameter to obtain the inlining behavior that you want. For instance, if the default setting for this parameter is leading to large functions being inlined and in turn generating large C code, you can tune the parameter in steps until you are satisfied with the size of generated code.

Default: 200

IntegrityChecks

Detect violations of memory integrity in code generated for MATLAB functions and stops execution with a diagnostic message.

Setting `IntegrityChecks` to `false` also disables the run-time stack.

Default: true

LaunchReport

Specify whether to automatically display HTML reports after code generation is complete or an error occurs.

Default: true

MATLABSourceComments

Include MATLAB source code as comments in the generated code.

Dependencies:

· `GenerateComments` enables this parameter.

Default: false

3-117

MemcpyThreshold

Specify the minimum number of bytes for the code generator to invoke the `memcpy` optimization or the `memset` optimization. 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. A `memcpy` or `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

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.

See "memcpy Optimization" and "memset Optimization".

Default: 64

Name

Name of code generation configuration object, specified as a character vector.

Default: `'MexCodeConfig'`

PostCodeGenCommand

Specify command to customize build processing after MEX function generation using `codegen`. Specify command as a character vector.

Default: `''`

PreserveVariableNames

Specify which variable names the code generator must preserve in the generated code.

Value	Description
`'None'`	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.

Default: `'None'`

ReportPotentialDifferences

Specify whether to report potential behavior differences between generated code and MATLAB code. If `ReportPotentialDifferences` is `true`, the code generation report has a tab that lists the potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

Default: `true`

ReservedNameArray

Enter a list of names that MATLAB Coder is not to use for naming functions or variables. Specify list as a character vector.

Default: `''`

ResponsivenessChecks

Enable responsiveness checks in code generated for MATLAB functions.

These checks enable periodic checks for Ctrl+C breaks in the generated code. Enabling responsiveness checks also enables graphics refreshing.

Caution: These checks are enabled by default for safety. Without these checks, the only way to end a long-running execution might be to terminate MATLAB.

Default: `true`

SaturateOnIntegerOverflow

Overflows saturate to either the minimum or maximum value that the data type can represent. Otherwise, the overflow behavior depends on your target C compiler. 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.

Default: `true`

StackUsageMax

Specify the maximum stack usage per application in bytes. Set a limit that is lower than the available stack size. Otherwise, a runtime stack overflow might occur. Overflows are detected and reported by the C compiler, not by `codegen`.

Default: `200000`

TargetLang

Specify the target language. Set to `'C'` to generate C code. Set to `C++` to generate C++ code. If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Default: `'C'`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Examples

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('mex')
```

3   Enable the code generation report.

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

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

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

Alternatives

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

See Also
codegen | coder | coder.extrinsic

coder.PrimitiveType class

Package: coder
Superclasses: coder.ArrayType

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. Name must appear inside single quotes (' '). 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 single 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) in the MATLAB documentation.

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

coder.ArrayType | `coder.typeof` | coder.Type | `coder.newtype` | `coder.resize` | `codegen`

Topics

Introduced in R2011a

coder.StructType class

Package: coder
Superclasses: coder.ArrayType

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.

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) in the MATLAB documentation.

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

coder.EnumType | coder.FiType | coder.Constant | coder.ArrayType | `coder.typeof` | `coder` | `coder.cstructname` | coder.Type | coder.PrimitiveType | `coder.newtype` | `coder.resize` | `codegen`

Topics

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

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
```

- "Generate Single-Precision MATLAB Code"
- "Generate Single-Precision C Code at the Command Line"

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

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) in the MATLAB documentation.

See Also

coder.EnumType | coder.FiType | coder.Constant | coder.ArrayType | `coder.typeof` | coder | coder.CellType | coder.StructType | coder.PrimitiveType | `coder.newtype` | `coder.resize` | codegen

Topics

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 single 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

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

See Also

See Also
coder.make.BuildItem | coder.make.BuildTool | coder.make.ToolchainInfo | coder.make.ToolchainInfo.getBuildConfiguration |

```
coder.make.ToolchainInfo.removeBuildConfiguration
| coder.make.ToolchainInfo.setBuildConfiguration |
coder.make.ToolchainInfo.setBuildConfigurationOption
```

Topics
"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

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Example

```
bi1 = coder.make.BuildItem('BuildItemMacroValue')


bi1 =

 Macro  : (empty)
 Value : BuildItemMacroValue

bi2 = coder.make.BuildItem('BIMV','BuildItemMacroValue')
```

```
bi2 =

 Macro  : BIMV
 Value : BuildItemMacroValue
```

See Also

See Also
```
coder.make.ToolchainInfo | coder.make.BuildTool |
coder.make.ToolchainInfo | coder.make.BuildTool.getCommand |
coder.make.BuildTool.setCommand | coder.make.BuildTool.setPath |
coder.make.BuildTool.addFileExtension
```

Topics

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: 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 3-152.

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

Assembler

C Compiler

C++ Compiler

Linker

Archiver

Download

Execute

BuilderApplication

BuildConfigurations

Faster Builds

Faster Runs

Debug

Assembler options

C Compiler options

C++ Compiler options

Linker options

Share Lib Linker options

Archiver options
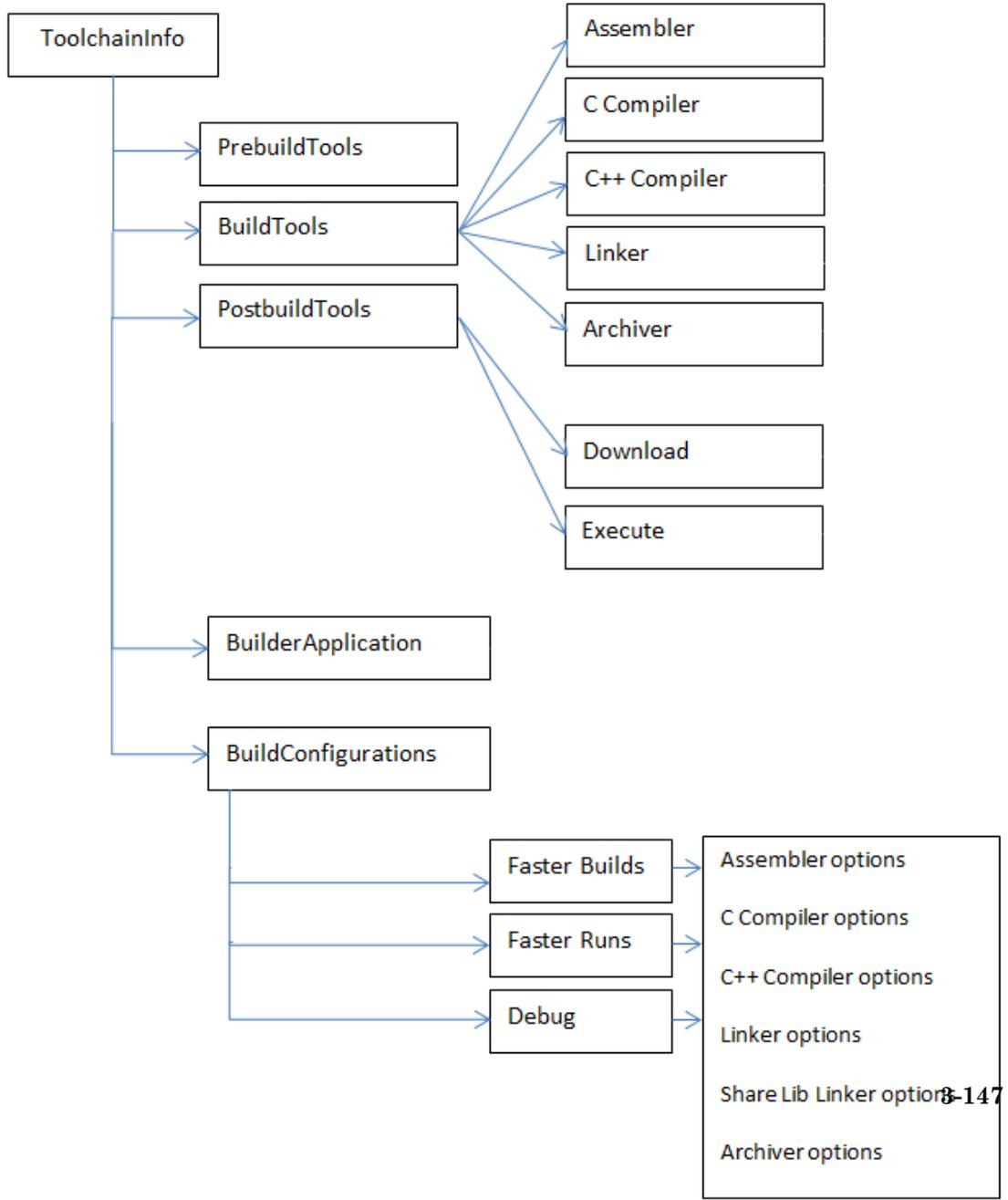
Construction

`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

Build tool name, specified as a character vector.

Data Types: `char`

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`:

- `coder.make.BuildTool.getCommand`
- `coder.make.BuildTool.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`:

- `coder.make.BuildTool.addDirective`
- `coder.make.BuildTool.getDirective`
- `coder.make.BuildTool.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`:

- `coder.make.BuildTool.addFileExtension`
- `coder.make.BuildTool.getFileExtension`
- `coder.make.BuildTool.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`.

- `coder.make.BuildTool.getName`
- `coder.make.BuildTool.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`:

- `coder.make.BuildTool.getPath`
- `coder.make.BuildTool.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

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Example

Get a Default Build Tool and Set Its Properties

The `intel_tc.m` file from 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: 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

```matlab
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'};
```

```matlab
% 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

% --------------------------------
```

```matlab
% [Input/Output]FileExtensions
% -------------------------------
if isfield(toolinfo,'InputFileExtensions')
    buildToolObj.InputFileExtensions = toolinfo.InputFileExtensions;
end
if isfield(toolinfo,'OutputFileExtensions')
    buildToolObj.OutputFileExtensions = toolinfo.OutputFileExtensions;
end
```

createBuildTool_2.m

```matlab
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)'};
```

3-155

```
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.getDef
% Set inlined commands for source to dependency tool
tc.InlinedCommands = ['define sourceToDep=', 10, ...
'$(foreach source, $(1), $(CC) $(CFLAGS) -E -MMD -MP -MF"$(notdir $(source:%.c=%.d))" -
'endef'];
% Set makefile includes
make = tc.BuilderApplication();
make.IncludeFiles = {'*.d'};
% Dependency File Generator for GCC-based toolchain
prebuildToolName = 'Dependency File Generator';
prebuildTool = tc.addPrebuildTool(prebuildToolName);
% Set command macro and value
prebuildTool.setCommand('SRC2DEP', '$(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_

% displays post-build tool properties
tc.getPostbuildTool('elf2hex converter')
```

Add Source, Includes, and Libraries for Build Tools

You can add sources, include paths, or libraries for build tools through the TOOLCHAIN_SRCS, TOOLCHAIN_INCS, and TOOLCHAIN_LIBS macros in the generated makefile. The toolchain generates these macro values from the Sources, IncludePaths, and Libraries fields of the build tool.

You can use the code in addSrcsIncsLibsForBuildToolToIntelToolchainInfo.m with the example. For example, apply this code after the *Registering a Toolchain* step and before the *Choosing the Toolchain* step. To view application of these build tool macros, generate code with the example and see the TOOLCHAIN_SRCS, TOOLCHAIN_INCS, and TOOLCHAIN_LIBS macros in the generated makefile, coderrand_rtw.mk.

The code in addSrcsIncsLibsForBuildToolToIntelToolchainInfo.m demonstrates this example for the GCC toolchain.

addSrcsIncsLibsForBuildToolToIntelToolchainInfo.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding sources, includes, and libraries for a build tool to Intel ToolchainInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Get toolchain info (created by Adding Custom Toolchain example) from registry
tc = coder.make.getToolchainInfoFromRegistry('Intel v12.1 | nmake makefile (64-bit Wind
% Add intrinsic macros
tc.addIntrinsicMacros('cvarsmd');
tc.setMacro('CVARSFLAG', '$(cvarsmd)');
% Get a build tool from toolchain object
bt = tc.getBuildTool('C Compiler');
% Add value for build tool sources macro
bt.Sources = {'myHardwareInit.c','myInterrupts.c'};
% Add value for build tool include paths macro
bt.IncludePaths = {'c:\work\libraries\','c:\vendor\libraries\'};
% Add value for build tool libraries macro
bt.Libraries = {'libiomp5md.lib'}; %add OpenMP library for OpenMP support…
% Updated toolchain object with build tool macros
tc.setBuildTool('C Compiler',bt);
% Save toolchain changes
save intel_tc tc
% Update registry
RTW.TargetRegistry.getInstance('reset')
```

addSrcsIncsLibsForBuildToolToGccToolchainInfo.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding sources, includes, and libraries for a build tool to gcc/g++ ToolchainInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Copy the GNU GCC ToolchainInfo object
tc = coder.make.getToolchainInfoFromRegistry('GNU gcc/g++ v4.4.x | gmake (64-bit Linux)
bt = tc.getBuildTool('C Compiler');
% Add value for build tool sources macro
bt.Sources = {'myHardwareInit.c','myInterrupts.c'};
% Add value for build tool include paths macro
bt.IncludePaths = {'/home/user/myTarget/include', '/home/vendor/drivers/include'};
% Add value for build tool libraries macro
bt.Libraries = {'-lm','-lrt'};
tc.setBuildTool('C Compiler',bt);
```

See Also

See Also
coder.make.ToolchainInfo | "Toolchain Definition File with Commentary"
| coder.make.BuildTool | coder.make.ToolchainInfo.addBuildTool
| coder.make.ToolchainInfo.getBuildTool |
coder.make.ToolchainInfo.removeBuildTool |
coder.make.ToolchainInfo.setBuildTool

Topics

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 `coder.make.getDefaultToolchain` and `coder.make.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
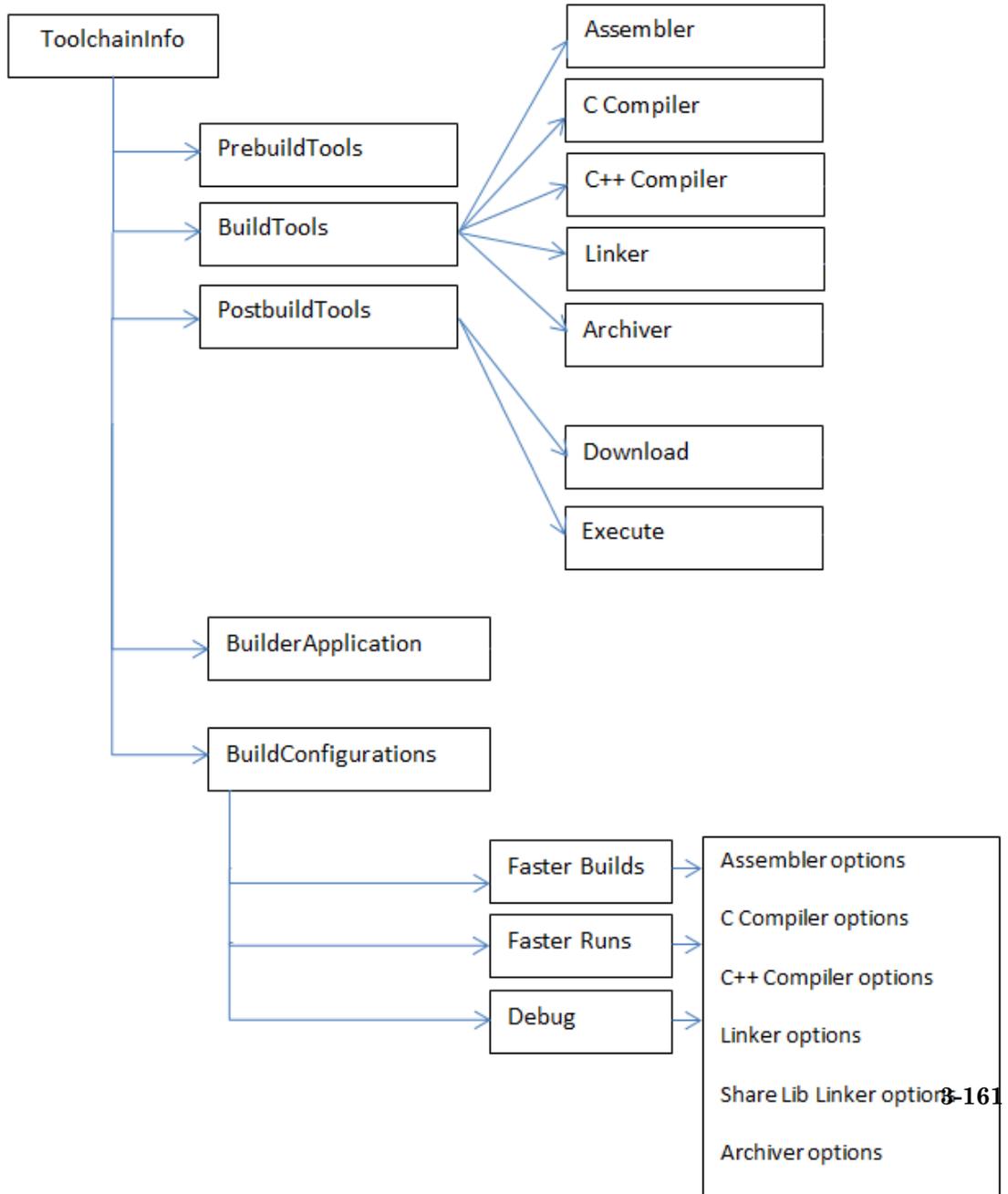
Construction

`h = coder.make.ToolchainInfo` creates a default ToolchainInfo 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 3-164 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 3-164 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 3-164 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 3-164 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 3-164 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 3-164 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 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 , 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`:

- `coder.make.ToolchainInfo.addAttribute`
- `coder.make.ToolchainInfo.getAttribute`
- `coder.make.ToolchainInfo.getAttributes`
- `coder.make.ToolchainInfo.isAttribute`
- `coder.make.ToolchainInfo.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 3-162.

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 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 , 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`:

- `coder.make.ToolchainInfo.getBuildConfiguration`
- `coder.make.ToolchainInfo.removeBuildConfiguration`
- `coder.make.ToolchainInfo.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 3-162.

The `BuildTool` objects ToolchainInfo can create based on the `SupportedLanguages` are:

- Assembler
- C Compiler
- C++ Compiler
- Linker
- Archiver

3-167

For example, the `intel_tc.m` file from , 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`:

- `coder.make.ToolchainInfo.addBuildTool`
- `coder.make.ToolchainInfo.getBuildTool`
- `coder.make.ToolchainInfo.removeBuildTool`
- `coder.make.ToolchainInfo.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 3-162.

For example, the `intel_tc.m` file from , 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 `coder.make.ToolchainInfo.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 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 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 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 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:

- coder.make.ToolchainInfo.addMacro
- coder.make.ToolchainInfo.getMacro
- coder.make.ToolchainInfo.removeMacro
- coder.make.ToolchainInfo.setMacro
- coder.make.ToolchainInfo.addIntrinsicMacros
- coder.make.ToolchainInfo.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 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.

3-175

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 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 example does not include the `PostbuildTools` property.

Use the following methods with this property:

- `coder.make.ToolchainInfo.addPostbuildTool`
- `coder.make.ToolchainInfo.getPostbuildTool`
- `coder.make.ToolchainInfo.removePostbuildTool`
- `coder.make.ToolchainInfo.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 example does not include the `PrebuildTools` property.

Use the following methods with this property:

- `coder.make.ToolchainInfo.addPrebuildTool`

3-177

- `coder.make.ToolchainInfo.getPrebuildTool`
- `coder.make.ToolchainInfo.removePrebuildTool`
- `coder.make.ToolchainInfo.setPrebuildTool`

Attributes:

```
GetAccess                              public
SetAccess                              public
```

`Revision` — Assign revision number to ToolchainInfo

Assign revision number to ToolchainInfo

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

3-179

      a   Download

      b   Execute

6   Shell Cleanup

7   MATLAB Cleanup

The 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 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.

If you do not specify a value for SupportedLanguages, 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.

To override the default, use a name-value pair to specify a value for SupportedLanguages when you initialize ToolchainInfo. 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 property does not have any related methods.

The example does not include the SupportedLanguages property.

Attributes:

```
GetAccess                              public
SetAccess                              public
```

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 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

See Also

See Also
```
coder.make.getDefaultToolchain |
coder.make.getToolchainInfoFromRegistry |
coder.make.BuildConfiguration | coder.make.BuildItem |
coder.make.BuildTool
```

Topics
"About coder.make.ToolchainInfo"

Tools — Alphabetical List

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries. Use the tool to explore and choose a library. If you develop a custom code replacement library, use the tool to verify table entries.

- 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 that the library contains. When you select a code replacement table, the viewer displays function and operator code replacement entries that are in that table.

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`).
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.

Field	Description
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`

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`

Field	Description
	`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

Field	Description
	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

```
crviewer('ARM Cortex-A')
```
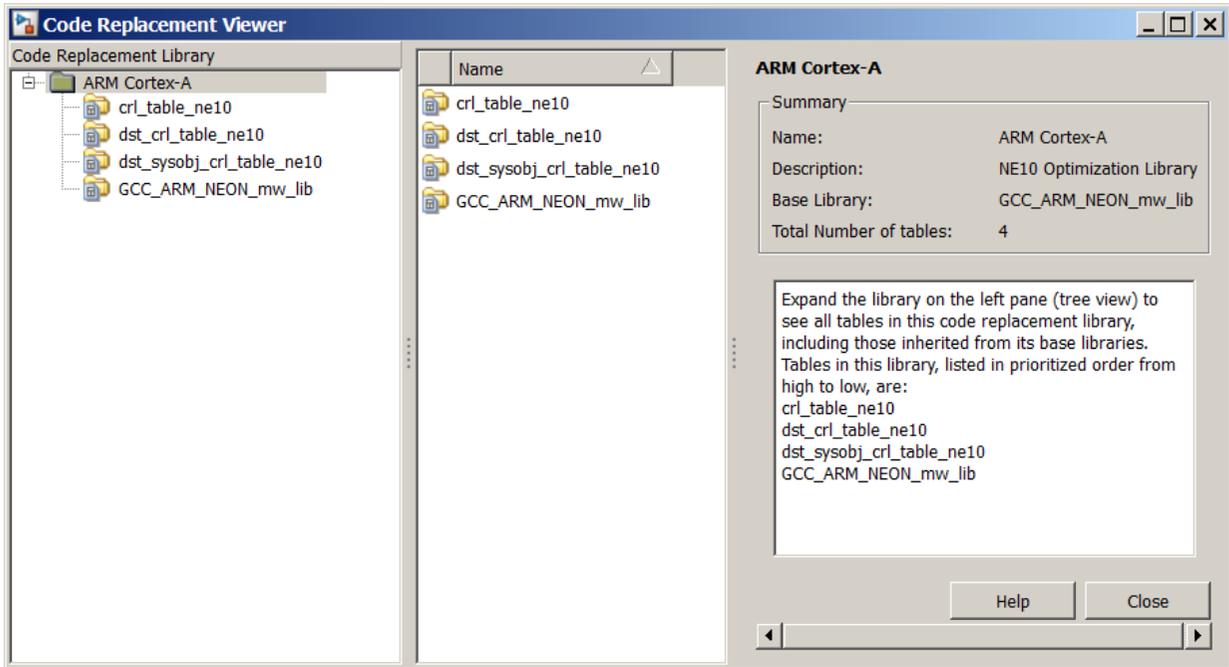
Display Contents of Code Replacement Table

```
crviewer(clr_table_ne10)
```
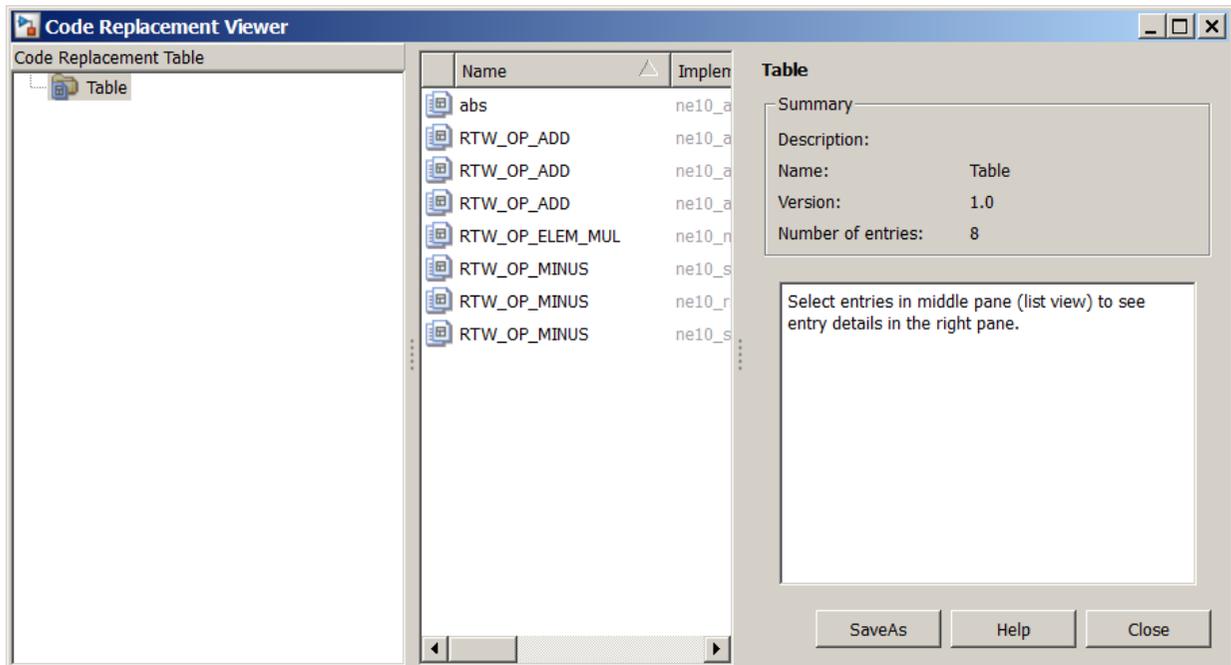
- "Choose a Code Replacement Library"

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. For example, 'GNU 99 extensions'.

crviewer(table) opens the Code Replacement Viewer and displays the contents of table, where table is a MATLAB file that defines code replacement tables. 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