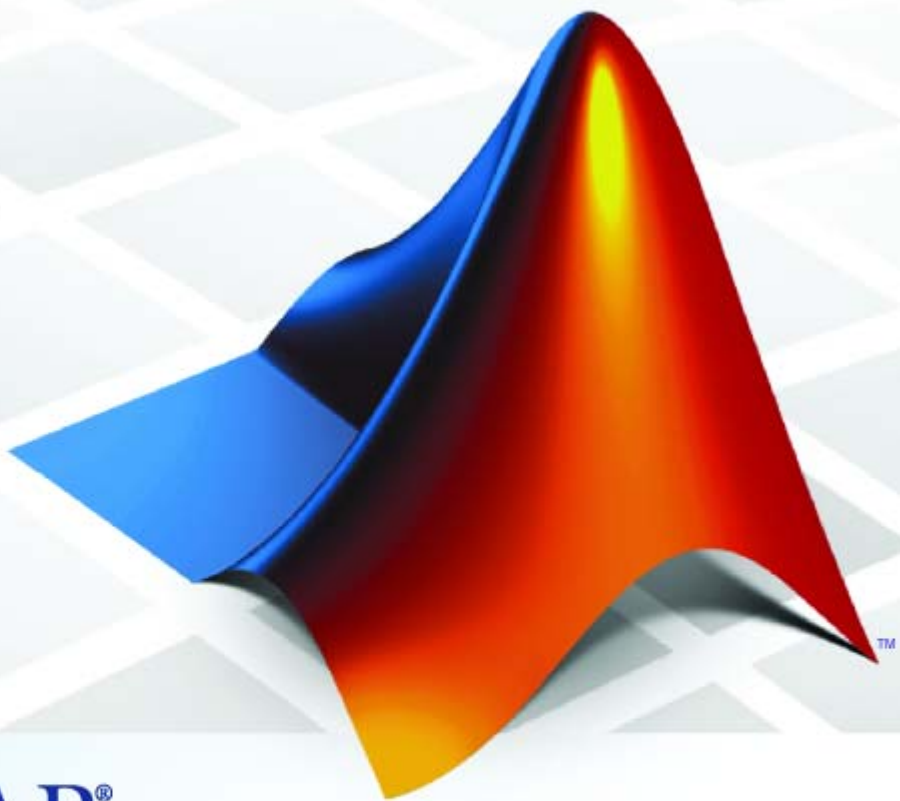


Embedded MATLAB™ Reference



MATLAB®

How to Contact The MathWorks



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



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Embedded MATLAB™ Reference

© COPYRIGHT 2008–2009 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

October 2008	Online only	New for Release 2008b
March 2009	Online only	Revised for Release 2009a
September 2009	Online only	Revised for Release 2009b

1 | Function Reference

| Index

Function Reference

Purpose Generate C-MEX code from M-code

Syntax `emlmex [-options] fun`

Description `emlmex` is a MATLAB® command that invokes Embedded MATLAB™ MEX. You issue the `emlmex` command from the MATLAB command prompt.

`emlmex [-options] fun` translates the M-file `fun.m` to a C-MEX file and generates all necessary wrapper files.

By default, `emlmex`:

- Converts the M-function `fun.m` to a C-MEX function
- Generates a platform-specific MEX-file in the current directory
- Stores generated files in the subdirectory `emcprj/mexfcn/fun/`

You can change the default behavior by specifying one or more compilation options as described in “Options” on page 1-2.

Options

You can specify one or more compilation options with each `emlmex` command. Use spaces to separate options and arguments. `emlmex` resolves options from left to right, so if you use conflicting options, the rightmost one prevails. Here is the list of options:

- “-d Specify Output Directory” on page 1-3
- “-eg Specify Input Properties by Example” on page 1-3
- “-F Specify Default fimath” on page 1-3
- “-g Compile C-MEX Function in Debug Mode” on page 1-4
- “-I Add Directories to Embedded MATLAB Path” on page 1-4
- “-N Specify Default Numeric Type” on page 1-4
- “-o Specify Output File Name” on page 1-5
- “-O Specify Compiler Optimization Option” on page 1-5

- “-report Generate Compilation Report” on page 1-5
- “-s Specify Compiler Options” on page 1-6
- “-? Display Help” on page 1-6

-d Specify Output Directory

`-d out_directory`

Store generated files in directory path specified by *out_directory*. If any directories on the path do not exist, emlmex creates them for you. *out_directory* can be an absolute path or relative path. If you do not specify an output directory, emlmex stores generated files in a default subdirectory called `emcprj/mexfcn/fun`.

-eg Specify Input Properties by Example

`-eg example_inputs`

Use the values in cell array *example_inputs* as sample inputs for defining the properties of the primary M-function inputs. The cell array should provide the same number and order of inputs as the primary function. See “Defining Input Properties by Example at the Command Line”.

-F Specify Default fimath

`-F fimath`

Use *fimath* as the default `fimath` object for all fixed-point inputs to the primary function. You can define the default value using the Fixed-Point Toolbox™ function `fimath`, as in this example:

```
emlmex -F fimath('OverflowMode','saturate','RoundMode','nearest') myFcn
```

emlmex uses the default value if you have not defined any other `fimath` property for the primary, fixed-point inputs, either by example (see “Defining Input Properties by Example at the Command Line”) or programmatically (see “Defining Input Properties Programmatically in the M-File”). If you do not define a default value, emlmex uses the MATLAB default `fimath` value.

-g Compile C-MEX Function in Debug Mode

Compile the C-MEX function in debug mode, with optimization turned off. If you do not specify `-g`, `emlmex` compiles the C-MEX function in optimized mode. You specify these modes using the `mex -setup` procedure described in “Building MEX-Files” in the MATLAB External Interfaces documentation.

-I Add Directories to Embedded MATLAB Path

`-I include_path`

Add `include_path` to the Embedded MATLAB path. By default, the Embedded MATLAB path consists of the current directory (`pwd`) and the Embedded MATLAB libraries directory. `emlmex` converts M-code to C-MEX code only if it finds the M-file on the Embedded MATLAB path. See “How the Embedded MATLAB Subset Resolves Function Calls”.

`emlmex` searches directories from left to right.

-N Specify Default Numeric Type

`-N numerictype`

Use `numerictype` as the default `numerictype` object for all fixed-point inputs to the primary function. You can define the default value using the Fixed-Point Toolbox function `numerictype`, as in this example:

```
emlmex -N numerictype(1,32,23) myFcn
```

This command specifies that the numeric type of all fixed-point inputs to the top-level function `myFcn` be signed (1), have a word length of 32, and have a fraction length of 23.

Embedded MATLAB MEX uses the default value if you have not specified any other numeric type for the primary, fixed-point inputs, either by example (see “Defining Input Properties by Example at the Command Line”) or programmatically (see “Defining Input Properties Programmatically in the M-File”). If you do not define a default value, then you must use one of the other methods to specify the numeric type of your primary, fixed-point inputs.

-o Specify Output File Name

-o output_file_name

Generate the final output file, the C-MEX function, with the base name *output_file_name*. Embedded MATLAB MEX automatically assigns C-MEX files a platform-specific extension (see “Naming Conventions”).

You can specify *output_file_name* as a file name or an existing path, with the following effects:

If you specify:	emlmex:
A file name	Copies the MEX-file to the current directory
An existing path	Generates the MEX-file in the directory specified by the path, but does not copy the MEX-file to the current directory
A path that does not exist	Generates an error

-O Specify Compiler Optimization Option

-O optimization_option

Specify compiler *optimization_option* with one of the following literals (no quotes):

Compiler Optimization Option	Action
<code>disable:inline</code>	Disable function inlining.
<code>enable:inline</code>	Enable function inlining (default).

-report Generate Compilation Report

Generate a compilation report. If this option is not specified, `emlmex` generates a report only if there are error or warning messages. For detailed information, see “Working with Compilation Reports”.

-s Specify Compiler Options

-s compilation_config_object

Generate C-MEX functions based on the properties of a compilation configuration object. When you specify conflicting configuration objects on the command line, the rightmost configuration object prevails. For detailed information, see “Setting C-MEX Compilation Options”.

If a compilation configuration object is not specified, Embedded MATLAB Coder uses default property values, as follows:

Defaults for emlcoder.CompilerOptions.

Property	Default
ConstantFoldingTimeout	10000
InlineThreshold	10
InlineThresholdMax200	200
InlineStackLimit	4000
SaturateOnIntegerOverflow	true
StackUsageMax	200000

-? Display Help

Display emlmex command help.

Examples

This section presents examples based on an M-file `emcrand.m`, described in “Sample M-File” on page 1-6.

Sample M-File

```
function r = emcrand(num)
assert(isa(num, 'double'));
persistent seeded;
if isempty(seeded)
    seeded = true;
    rand('seed', num);
```

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

Converting M-Function to C-MEX Function

```
emlmex emcrand
```

Generates a C-MEX function. Places the C-MEX function and other supporting files in a subdirectory called `emcprj/mexfcn/emcrand`, the default location. `emlmex` uses the name of the M-function as the root name for the generated files and creates a platform-specific extension for the C-MEX file, as described in “Naming Conventions”.

Specifying Custom Name for C-MEX File

```
emlmex -o emcrandmx emcrand
```

Uses `emcrandmx` as the root name of the C-MEX file, but uses `emcrand` as the root name for all other generated files. Generates all files to the default directory `emcprj/mexfcn/emcrand`, but also makes a copy of the C-MEX file in the current directory.

Specifying Custom File Name as Path

```
emlmex -o mydir/emcrandx emcrand
```

Generates all files in an existing subdirectory called `mydir`, using `emcrandx` as the root name of the C-MEX file. When the argument is a path, `emlmex` does not copy the C-MEX file to the current directory.

Specifying Custom Output Directory for C-MEX File

```
emlmex -d mydir emcrand
```

Generates all files in the subdirectory `mydir` with the M-function name as the root name for all files.

Specifying Primary Function Input Properties by Example

Currently, the M-function `emcrand` (described in “Sample M-File” on page 1-6) uses the `assert` function to specify that its input `num` is a real double scalar, as follows:

```
assert(isa(num, 'double'));
```

Note For information about using `assert` to specify input properties for `emlmex`, see “Defining Input Properties Programmatically in the M-File”.

Suppose you instead want to specify the primary function input properties by example at the command line. Remove the `assert` call from the M-code and enter this command:

```
emlmex -eg {0} emcrand
```

The value in the cell array `{0}` is a real double scalar, exemplifying the properties that you want to specify for input `num`.

Purpose Allow compilation of protected Embedded MATLAB compliant M-code

Syntax `eml.allowpcode('plain')`

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

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

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

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

`eml.allowpcode` has no effect in MATLAB code; it applies to the Embedded MATLAB subset only.

Examples Generate optimized embeddable code from protected M-code:

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

```
function out = p_abs(in) %#eml
% The directive %#eml declares the function
% to be Embedded MATLAB compliant
eml.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.

eml.allowpcode

- 3 Generate a C-MEX function for `p_abs.p`, using the `-eg` option to specify the size, class, and complexity of the input parameter. At the MATLAB prompt, enter:

```
emlmex p_abs -eg { int32(0) }
```

`emlmex` generates a C-MEX function in the current folder.

- 4 Generate embeddable C code for `p_abs.p` (requires a Real-Time Workshop® license). At the MATLAB prompt, enter:

```
emlc p_abs -T rtw:lib -eg { int32(0) };
```

`emlc` generates C library code in the `emcprj\rtwlib\p_abs` folder.

See Also

`pcode` | `emlmex` | `emlc`

How To

- “Adding the Compilation Directive `%#eml`”

Purpose Call external C function

Syntax

```
eml.ceval('cfun_name')
eml.ceval('cfun_name', cfun_arguments)
cfun_return = eml.ceval('cfun_name')
cfun_return = eml.ceval('cfun_name', cfun_arguments)
```

Description `eml.ceval('cfun_name')` executes the external C function specified by the quoted string `cfun_name`. Define `cfun_name` in an external C source file or library.

`eml.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 = eml.ceval('cfun_name')` executes `cfun_name` and returns a single scalar value, `cfun_return`, corresponding to the value that the C function returns in the return statement. To be consistent with C, `eml.ceval` can return only a scalar value.

`cfun_return = eml.ceval('cfun_name', cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments` and returns `cfun_return`.

To allow the Embedded MATLAB subset to infer the data type of return values and output arguments, specify their type, size, and complexity before calling `eml.ceval`.

By default, `eml.ceval` passes arguments by value to the C function whenever C supports passing arguments by value. To make `eml.ceval` pass arguments by reference, use the constructs `eml.ref`, `eml.rref`, and `eml.wref`. If C does not support passing arguments by value, for example, if the argument is an array, `eml.ceval` passes arguments by reference. In this case, if you do not use the `eml.ref`, `eml.rref`, and `eml.wref` constructs, Embedded MATLAB might introduce a copy of the argument in the generated code to enforce MATLAB semantics for arrays.

Use `eml.ceval` only in Embedded MATLAB code that you have compiled with `emlmex` or `emlc`. `eml.ceval` generates an error in uncompiled M-code. Use `eml.target` to determine if the MATLAB

function is executing in MATLAB. If it is, do not use `eml.ceval` to call the C function. Instead, call the MATLAB version of the C function.

Examples

Call a C function `foo(u)` from an Embedded MATLAB function:

- 1 Create a C header file `foo.h` for a function `foo` that takes two input parameters of type `real_T` and returns a value of type `int32_T`.

```
#include <tmwtypes.h>

int32_T foo(real_T in1, real_T in2);
```

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

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

int32_T foo(real_T in1, real_T in2)
{
    return in1 + in2;
}
```

- 3 Write an M-function `callfoo` that calls `foo` using `eml.ceval`.

```
function y = callfoo %#eml
% The directive %#eml declares the function
% to be Embedded MATLAB compliant
y = int32(0); % Constrain the return type to int32_T
if isempty(eml.target)
    % Executing in MATLAB, call MATLAB equivalent of
    % C function foo
    y = 10 + 20;
else
    % Executing in Embedded MATLAB, call C function foo
    y = eml.ceval('foo', 10, 20);
end
```

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

```
emlc -T rtw:lib callfoo foo.c foo.h
```

`emlc` generates C code in the `emcprj\rtwlib\callmfoo` subfolder.

```
int32_T callfoo(void)
{
    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`.

- 5** Cast the input arguments to specify their type explicitly.

```
y = eml.ceval('foo', int32(10), int32(20));
```

The generated code is:

```
int32_T callfoo(void)
{
    return foo(10, 20);
}
```

Call a C library function from M-code:

- 1** Write an M-function `absval`.

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

- 2 Generate the C library for `absval.m`, using the `-eg` option to specify the size, type, and complexity of the input parameter. `emlc` creates the library `absval.lib` and header file `absval.h` in the folder `/emcprj/rtwlib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the same folder.

```
emlc -T rtw:lib absval -eg {0.0}
```

- 3 Write an M-function to call the generated C library functions using `eml.ceval`.

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

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

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

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

```
emlc -T mex callabsval emcprj/rtwlib/absval/absval.lib...
emcprj/rtwlib/absval/absval.h
```

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

```
callabsval
```

See Also

[eml.ref](#) | [eml.rref](#) | [eml.wref](#) | [eml.target](#) | [emlmex](#) | [emlc](#)

Tutorials

- “Returning Multiple Values from C Functions”
- “Calling an External C Sort Function”

How To

- “Adding the Compilation Directive `%#eml`”
- “Calling C Functions from the Embedded MATLAB Subset”
- “Declaring Variables”

eml.cstructname

Purpose Specify structure name in generated code

Syntax `eml.cstructname(structVar, 'structName')`
`eml.cstructname(structVar, 'structName', 'extern')`

Description `eml.cstructname(structVar, 'structName')` allows you to specify the name of a structure in generated code. `structVar` is the structure variable. `structName` specifies the name to use for the structure.

`eml.cstructname(structVar, 'structName', 'extern')` declares an externally defined structure. The Embedded MATLAB subset does not generate the definition of the structure type; provide it in a custom include file.

You must call `eml.cstructname` before the first use of the structure variable in your function.

`eml.cstructname` has no effect in MATLAB code; it applies to the Embedded MATLAB subset only. Using `eml.cstructname` at the MATLAB command line and then calling `emlc` does not assign a name to a structure in the generated code.

Examples Apply `eml.cstructname` to top-level inputs:

- 1 Write an Embedded MATLAB compliant M-function `topfun` that assigns the name `MyStruct` to its input parameter.

```
function y = topfun(x) %#eml
% Instruct Embedded MATLAB to assign the name 'MyStruct'
% to the input variable
eml.cstructname(x, 'MyStruct');
y = x;
```

- 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 `-eg` option to specify that the input parameter is a structure.

```
emlc -eg { s } topfun.m
```

`emlc` generates a MEX function in the default folder `emcprj\mexfcn\topfun`. The structure definition is in `topfun_types.h` in this folder.

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

Assign the name `MyStruct` to the structure `structVar` and pass the structure to a C function `use_struct`:

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

```
#include <tmwtypes.h>

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

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

- 2 Write the C function `use_struct.c`.

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

eml.cstructname

```
#include "use_struct.h"

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

- 3** Write an Embedded MATLAB compliant M-function `m_use_struct` that declares a structure, assigns the name `MyStruct` to it, and then calls the C function `use_struct` using `eml.ceval`.

```
function m_use_struct %#eml
% The directive %#eml declares the function
% to be Embedded MATLAB compliant
% Declare a MATLAB structure
structVar.s1 = 1;
structVar.s2 = 2;

% Instruct Embedded MATLAB to assign the name MyStruct
% to the structure variable. extern indicates this is
% an externally defined structure.
eml.cstructname(structVar, 'MyStruct', 'extern');

% Call the C function use_struct. The type of structVar
% matches the signature of use_struct.
% Use eml.rref to pass the the variable structVar by
% reference as a read-only input to the external C
% function use_struct
eml.ceval('use_struct', eml.rref(structVar));
```

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

```
emlc -T rtw:lib m_use_struct use_struct.c use_struct.h
```

emlc generates C code in the default folder `emcprj\rtwlib\m_use_struct`. The generated header file `m_use_struct_types.h` in this folder contains no definition of the structure `MyStruct` because `MyStruct` is an external type.

See Also

[eml.ceval](#) | [eml.rref](#) | [emlc](#)

How To

- “Adding the Compilation Directive `%#eml`”
- “Working with Structures”

eml.extrinsic

Purpose

Declare extrinsic function or functions

Syntax

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

Description

`eml.extrinsic` declares extrinsic functions. An extrinsic function is an M-function on the MATLAB path that Embedded MATLAB functions dispatch to MATLAB for execution. The Embedded MATLAB subset does not compile or generate code for extrinsic functions, provided they do not affect execution of the host function; otherwise, Embedded MATLAB issues compilation errors.

`eml.extrinsic` has no effect in MATLAB code; it applies to the Embedded MATLAB subset only.

Example

The following code declares the MATLAB functions `patch` and `axis` extrinsic in the Embedded MATLAB subfunction `create_plot`:

```
function c = pythagoras(a,b,color) %#eml  
% 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 and axis as extrinsic  
  
eml.extrinsic('patch', 'axis');
```

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

By declaring these functions extrinsic, you instruct Embedded MATLAB not to compile or generate code for `patch` and `axis`, but instead dispatch them to MATLAB for execution.

eml.inline

Purpose Control inlining in generated code

Syntax `eml.inline('always')`
`eml.inline('never')`
`eml.inline('default')`

Description `eml.inline('always')` forces inlining of the current function in generated code.

`eml.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 M-source code and the generated C code.

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

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

Place the `eml.inline` directive inside the function to which it applies.

- Examples**
- “Preventing Function Inlining” on page 1-22
 - “Using `eml.inline` In Control Flow Statements” on page 1-23

Preventing Function Inlining

In this simple example, function `foo` will not be inlined in the generated code.

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

Using eml.inline In Control Flow Statements

You can use `eml.inline` in control flow code. When the Embedded MATLAB subset detects contradictory `eml.inline` directives, it 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 `safe_division` manually controls inlining based on whether it performs scalar division or vector division.

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

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

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

y = dividend / divisor;
```

eml.nullcopy

Purpose Declare uninitialized variables

Syntax `X = eml.nullcopy(A)`

Description `X = eml.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.

Use With Caution

Use this function with caution. See “Rules for Declaring Uninitialized Variables” in the Embedded MATLAB User’s Guide documentation.

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

Using `eml.nullcopy` with `zeros` lets you specify the size of vector `X` without initializing each element to zero.

See Also “Declaring Uninitialized Variables”

Purpose Declare variable in generated code

Syntax `y = eml.opaque(type, [value]);`

Arguments

y

Specifies the variable declared in the generated code.

type

Specifies the data type. Specify a C type defined in the user include file to avoid compilation errors. The C type provided must support assignment in C.

Note Arrays in C cannot be directly assigned, for example, the type declaration `int[50]` is not valid.

value (optional)

Specifies the data value declaration. Specify a C expression not dependent on Embedded MATLAB variables or functions.

If you do not define an initial value, you must initialize the value before using the data. Using the data without prior assignment can lead to compilation errors.

Description

`y = eml.opaque(type, [value]);` declares data of the type *type*, and the optional initial value *value*. `eml.opaque` allows you to manipulate C data that the Embedded MATLAB subset does not recognize. *type* and *value* are treated as strings by Embedded MATLAB functions. Data initialized with `eml.opaque` can be:

- Assigned to each other as long as their types are identical
- An argument to `eml.rref`, `eml.wref`, or `eml.ref`
- An input or output argument to `eml.ceval`

- An input or output argument to a user-written Embedded MATLAB function
- An input to a limited subset of Embedded MATLAB library functions

`eml.opaque` declares the type of a variable; it does not instantiate the variable. You can instantiate a variable using `eml.ceval` after declaring the variable type with `eml.opaque`. For example:

```
% Declare fp1 of type FILE *
fp1 = eml.opaque('FILE *');

%Create the variable fp1
fp1 = eml.ceval('fopen', cstring('filetest.m'), cstring('r'));
```

Example

The following example uses `eml.opaque` to declare a variable `f` as a `FILE *` type.

```
% This example returns its own source code by using
% fopen/fread/fclose.
function buffer = filetest

% Declare 'f' as an opaque type 'FILE *'
f = eml.opaque('FILE *', 'NULL');
% Open file in binary mode
f = eml.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, Embedded MATLAB converts all constant values
    % to doubles, so explicit type conversion to int32 is inserted.
    n = eml.ceval('fread', eml.ref(buffer(i)), int32(1), ...
        int32(numel(buffer)), f);
    i = i + n;
```

```
end
eml.ceval('fclose',f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB string
function y = cstring(x)
    y = [x char(0)];

% Remove all character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
j = 1;
for i = 1:numel(buffer)
    if buffer(i) ~= char(13)
        buffer(j) = buffer(i);
        j = j + 1;
    end
end
buffer(i) = 0;
```

See Also

`eml.ceval`, `eml.ref`, `eml.rref`, `eml.wref`

Purpose	Pass argument by reference as read input or write output
Syntax	<code>[y =] eml.ceval('function_name', eml.ref(arg), ... u_n)</code>
Arguments	<i>arg</i> Variable passed by reference as an input or an output to the external C function called in <code>eml.ceval</code> .
Description	<code>[y =] eml.ceval('function_name', eml.ref(arg), ... u_n)</code> passes the variable <i>arg</i> by reference as an input or an output to the external C function called in <code>eml.ceval</code> . You add <code>eml.ref</code> inside <code>eml.ceval</code> as an argument to <i>function_name</i> . The argument list can contain multiple <code>eml.ref</code> constructs. Add a separate <code>eml.ref</code> construct for each argument that you want to pass by reference to <i>function_name</i> . Only use <code>eml.ref</code> in Embedded MATLAB code that you have compiled with <code>emlmex</code> or <code>emlc</code> . <code>eml.ref</code> generates an error in uncompiled M-code.

Example

In the following example, an Embedded MATLAB function `fcn` has a single input `u` and a single output `y`. `fcn` calls a C function `my_fcn`, passing `u` by reference as an input. The value of output `y` is passed to `fcn` by the C function through its return statement.

Here is the Embedded MATLAB function code:

```
function y = fcn(u)

y = 0; %Constrain return type to double
y = eml.ceval('my_fcn', eml.ref(u));
```

The corresponding C function prototype looks like this:

```
real_T my_fcn(real_T *a)
```

In this example, the Embedded MATLAB subset infers the type of the input `u` from its definition in the parent model.

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

Embedded MATLAB cannot infer the type of the output `y`, so you must set it explicitly—in this case to a constant value `0` whose type defaults to `double`, matching the C type `real_T`. For a list of type mappings, see “Mapping MATLAB Types to C”.

See Also

`eml.ceval`, `eml.rref`, `eml.wref`

eml.rref

Purpose Pass argument by reference as read-only input

Syntax `[y =] eml.ceval('function_name', eml.rref(argI), ... un)`

Arguments *argI*
Variable passed by reference as a *read-only* input to the external C function called in `eml.ceval`.

Description `[y =] eml.ceval('function_name', eml.rref(argI), ... un)` passes the variable *argI* by reference as a *read-only* input to the external C function called in `eml.ceval`. You add `eml.rref` inside `eml.ceval` as an argument to *function_name*. The argument list can contain multiple `eml.rref` constructs. Add a separate `eml.rref` construct for each read-only argument that you want to pass by reference to *function_name*.

Caution

Embedded MATLAB assumes that a variable passed by `eml.rref` is *read-only* and optimizes the code accordingly. Consequently, the C function must not write to the variable or results can be unpredictable.

Only use `eml.rref` in Embedded MATLAB code that you have compiled with `emlmex` or `emlc`. `eml.rref` generates an error in uncompiled M-code.

Example In the following example, an Embedded MATLAB function `fcn` has a single input `u` and a single output `y`. `fcn` calls a C function `foo`, passing `u` by reference as a read-only input. The value of output `y` is passed to `fcn` by the C function through its `return` statement.

Here is the Embedded MATLAB function code:

```
function y = fcn(u)

y = 0; %Constrain return type to double
y = eml.ceval('foo', eml.rref(u));
```

The corresponding C function prototype looks like this:

```
real_T foo(real_T *a)
```

In this example, Embedded MATLAB infers the type of the input `u` from its definition in the parent model.

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

Embedded MATLAB cannot infer the type of the output `y`, so you must set it explicitly—in this case to a constant value 0 whose type defaults to `double`, matching the C type `real_T`. For a list of type mappings, see “Mapping MATLAB Types to C”.

See Also

`eml.ceval`, `eml.opaque`, `eml.ref`, `eml.wref`

eml.target

Purpose Determine Embedded MATLAB code generation target

Syntax [y =] eml.target

Description [y =] eml.target returns a string representing the Embedded MATLAB code generation target.

String	Description
' '	Function is executing in MATLAB
'rtw'	Real-Time Workshop target
'sfun'	S-function target (Simulation target)
'mex'	MEX-function target
'hdl'	Stateflow® HDL Coder target

Example Use eml.target to parameterize your Embedded MATLAB functions that use custom C code so that they work in MATLAB or MEX.

```
if isempty(eml.target)
    % running in MATLAB
else
    % running in Embedded MATLAB
end
```

See Also eml.ceval

Purpose

Copy body of for-loop in generated code for each iteration

Syntax

```
for i = eml.unroll(range)
for i = eml.unroll(range, flag)
```

Description

for *i* = eml.unroll(*range*) copies the body of a for-loop (unrolls a for-loop) in generated code for each iteration specified by the bounds in *range*. *i* is the loop counter variable.

for *i* = eml.unroll(*range*, *flag*) unrolls a for-loop as specified in *range* if *flag* is true.

You must use eml.unroll in a for-loop header. eml.unroll modifies the generated code, but does not affect the computed results.

eml.unroll must be able to evaluate the bounds of the for-loop at compile time. The number of iterations cannot exceed 1024; unrolling large loops can increase compile time significantly and generate inefficient code

This function has no effect in MATLAB code; it applies to the Embedded MATLAB subset only.

Inputs

flag

Boolean expression that indicates whether to unroll the for-loop:

true	Unroll the for-loop
------	---------------------

false	Do not unroll the for-loop
-------	----------------------------

range

Specifies the bounds of the for-loop iteration:

<code>init_val : end_val</code>	Iterate from <code>init_val</code> to <code>end_val</code> , using an increment of 1
<code>init_val : step_val : end_val</code>	Iterate from <code>init_val</code> to <code>end_val</code> , using <code>step_val</code> as an increment if positive or as a decrement if negative
Matrix variable	Iterate for a number of times equal to the number of columns in the matrix

Examples

Limit the number of times to copy the body of a for-loop in generated code:

- 1 Write an Embedded MATLAB compliant function `getrand(n)` that uses a for-loop to generate a vector of length `n` and assign random numbers to specific elements. Also, add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the for-loop in generated code.

```
function [y1, y2] = test_unroll() %#eml
% The directive %#eml declares the function
% to be Embedded MATLAB compliant
% Trigger flag variable = true
y1 = getrand(8);
% Trigger flag variable = false
y2 = getrand(50);

function y = getrand(n)
% Turn off inlining to make
% generated code easier to read
eml.inline('never');

% Set flag variable dounroll to repeat loop body
```

```

% only for fewer than 10 iterations
dounroll = n < 10;
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
for i = eml.unroll(1:2:n, dounroll)
    if (i > 2) && (i < n-2)
        y(i) = rand();
    end;
end;
% Loop body ends

```

- 2** Generate C library code for `test_unroll` in the default output directory `emcprj/rtwlib/test_unroll`:

```
emlc -T rtw:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the for-loop (unrolls the loop) because the number of iterations is less than 10:

```

static void m_getrand(real_T eml_y[8])
{
    int32_T eml_i0;
    for(eml_i0 = 0; eml_i0 < 8; eml_i0++) {
        eml_y[eml_i0] = 0.0;
    }
    /* Loop body begins */
    eml_y[2] = m_rand();
    eml_y[4] = m_rand();
    /* Loop body ends */
}

```

The generated C code for `getrand(50)` does not unroll the for-loop because the number of iterations is greater than 10:

```
static void m_b_getrand(real_T eml_y[50])
```

```
{
  int32_T eml_i;
  for(eml_i = 0; eml_i < 50; eml_i++) {
    eml_y[eml_i] = 0.0;
  }
  /* Loop body begins */
  for(eml_i = 0; eml_i < 50; eml_i += 2) {
    if((eml_i + 1 > 2) && (eml_i + 1 < 48)) {
      eml_y[eml_i] = m_rand();
    }
  }
  /* Loop body ends */
}
```

See Also

[eml.inline](#) | [eml.nullcopy](#) | [for](#)

How To

- “Writing Efficient Code”
- “Using Arrays as Indices”
- “Declaring Variables By Assignment”

Purpose	Declare variable-size data
Syntax	<pre>eml.varsize('var₁', 'var₂', ...) eml.varsize('var₁', 'var₂', ..., ubound) eml.varsize('var₁', 'var₂', ..., ubound, dims) eml.varsize('var₁', 'var₂', ..., [], dims)</pre>
Description	<p><code>eml.varsize('var₁', 'var₂', ...)</code> declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each <code>'var_n'</code> must be a quoted string that represents a variable, or structure field. If the structure field is a structure array, use colon (<code>:</code>) as the index expression, indicating that all elements of the array are variable sized. For example, the expression <code>eml.varsize('data(:).A')</code> declares that the field <code>A</code> inside each element of <code>data</code> is variable sized.</p> <p><code>eml.varsize('var₁', 'var₂', ..., ubound)</code> declares one or more variables as variable-size data with an explicit upper bound specified in <code>ubound</code>. The argument <code>ubound</code> must be a constant, integer-valued vector of upper bound sizes for every dimension of each <code>'var_n'</code>. If you specify more than one <code>'var_n'</code>, each variable must have the same number of dimensions.</p> <p><code>eml.varsize('var₁', 'var₂', ..., ubound, dims)</code> declares one or more variables as variable-sized with an explicit upper bound and a mix of fixed and varying dimensions specified in <code>dims</code>. The argument <code>dims</code> is a logical vector, or double vector containing only zeros and ones. Dimensions that correspond to zeros or <code>false</code> in <code>dims</code> have fixed size; dimensions that correspond to ones or <code>true</code> vary in size. If you specify more than one variable, each fixed dimension must have the same value across all <code>'var_n'</code>.</p> <p><code>eml.varsize('var₁', 'var₂', ..., [], dims)</code> declares one or more variables as variable-sized with a mix of fixed and varying dimensions. The empty vector <code>[]</code> means that you do not specify an explicit upper bound.</p> <p>When you do <i>not</i> specify <code>ubound</code>, Embedded MATLAB technology computes the upper bound for each <code>'var_n'</code>.</p>

When you do *not* specify *dims*, Embedded MATLAB assumes that all dimensions are variable except the singleton ones. A singleton dimension is any dimension for which $\text{size}(A, \text{dim}) = 1$.

You must add the `eml. varsize` declaration before each ' var_n ' is used (read). You may add the declaration before the first assignment to each ' var_n '.

This function has no effect in MATLAB code; it applies to the Embedded MATLAB subset only.

Examples

Develop a simple stack that varies in size up to 32 elements as you push and pop data at runtime.

- 1 Write primary function `test_stack` to issue commands for pushing data on and popping data from a stack. Write subfunction `stack` to execute the push and pop commands.

```
function test_stack %#eml
    % The directive %#eml declares the function
    % to be Embedded MATLAB compliant
    stack('init', 32);
    for i = 1 : 20
        stack('push', i);
    end
    for i = 1 : 10
        value = stack('pop');
        % Display popped value
        value
    end
end

function y = stack(command, varargin)
    persistent data;
    if isempty(data)
        data = ones(1,0);
    end
    y = 0;
```

```

switch (command)
case {'init'}
    eml.varsizes('data', [1, varargin{1}], [0 1]);
    data = ones(1,0);
case {'pop'}
    y = data(1);
    data = data(2:size(data, 2));
case {'push'}
    % The %#ok comment below indicates that use of
    % variable-size variable is intentional.
    data = [varargin{1}, data]; %#ok<EMGRO>
otherwise
    assert(false, ['Wrong command: ', command]);
end
end
end

```

The variable `data` is the stack. The statement `eml.varsizes('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

2 Generate a MEX function for `test_stack`:

```
emlmex test_stack
```

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

3 Run `test_stack` to get these results:

```
value =
    20
```

```
value =
    19
```

```
value =
```

eml.varsize

```
18
value =
17
value =
16
value =
15
value =
14
value =
13
value =
12
value =
11
```

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

Declare a variable-size structure field.

- 1 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() %#eml

d = struct('values', zeros(1,0), 'color', 0);
data = repmat(d, [3 3]);
eml.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 `eml.varsizes('data(:).values')` marks as variable-sized the field `values` inside each element of the matrix `data`.

2 Generate a MEX function for `struct_example`:

```
emlmex struct_example
```

3 Run `struct_example`.

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

Alternatives

You can use the `assert` function to constrain an upper bound within a range of values, such as when growing a variable in a loop.

See Also

`assert` | `emlmex` | `size` | `varargin`

How To

- “Generating Code for Variable-Size Data”
-
- “Working with the Embedded MATLAB Subset”
- “Adding the Compilation Directive `%#eml`”

eml.wref

- Purpose** Pass argument by reference as write-only output
- Syntax** `[y =] eml.ceval('function_name', eml.wref(arg0), ... u_n);`
- Arguments** `arg0`
Variable passed by reference as a *write-only* output to the external C function called in `eml.ceval`.
- Description** `[y =] eml.ceval('function_name', eml.wref(arg0), ... u_n);` passes the variable `arg0` by reference as a *write-only* output to the external C function called in `eml.ceval`. You add `eml.wref` inside `eml.ceval` as an argument to `function_name`. The argument list can contain multiple `eml.wref` constructs. Add a separate `eml.wref` construct for each write-only argument that you want to pass by reference to `function_name`.

Caution

The Embedded MATLAB subset assumes that a variable passed by `eml.wref` is *write-only* and optimizes the code accordingly. Consequently, the C function must write to the variable. If the variable is a vector or matrix, the C function must write to *every* element of the variable. Otherwise, results are unpredictable.

Only use `eml.wref` in Embedded MATLAB code that you have compiled with `emlmex` or `emlc`. `eml.wref` generates an error in uncompiled M-code.

Example

In the following example, an Embedded MATLAB function `fcn` has a single input `u` and a single output `y`, a 5-by-10 matrix. `fcn` calls a C function `init` to initialize the matrix, passing `y` by reference as a write-only output. Here is the Embedded MATLAB function code:

```
function y = fcn(u)
```

```
y = zeros(5,10,'int8'); %Constrain output to an int8 matrix
eml.ceval('init', eml.wref(y));
```

The corresponding C function prototype looks like this:

```
void init(int8_T *x);
```

In this example:

- Although the C function is void, `eml.wref` allows it to access, modify, and return a matrix to the Embedded MATLAB function.
- The C function prototype defines the output as a pointer because it is passed by reference.
- The Embedded MATLAB subset cannot infer the type of the output `y`, so you must set it explicitly—in this case to an `int8` matrix, matching the C type `int8_T`. For a list of type mappings, see “Mapping MATLAB Types to C”.
- Embedded MATLAB collapses matrices to a single dimension in the generated C code.

See Also

`eml.ceval`, `eml.ref`, `eml.rref`

C

Controlling inlining in generated code 1-22

D

Declaring extrinsic functions 1-20

Declaring uninitialized variables 1-24

Declaring variables in generated code 1-25

Determining code generation target 1-32

E

eml.extrinsic function 1-20

eml.inline function 1-22

eml.nullcopy function 1-24

eml.opaque function 1-25

eml.ref function 1-28

eml.rref function 1-30

eml.target function 1-32

eml.wref function 1-42

emlmex function 1-2

G

Generating C-MEX code from M-code 1-2

P

Passing arguments by reference as read input
or write output 1-28

Passing arguments by reference as read-only
input 1-30

Passing arguments by reference as write-only
output 1-42