Embedded MATLAB™
User's Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

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™ User's Guide

Revision History

March 2007	Online only	New for Release 2007a
September 2007	Online only	Updated for Release 2007b

Contents

1 Working with Embedded MATLAB

2 Working with Embedded MATLAB MEX

3 Calling C Functions from Embedded MATLAB

4 Functions — Alphabetical List

Index

Working with Embedded MATLAB

What Is Embedded MATLAB?

In this section...

A Subset of MATLAB

Embedded MATLAB is a subset of the MATLAB language that supports efficient code generation for deployment in embedded systems and acceleration of fixed-point algorithms.

Embedded MATLAB Inference Engine

Embedded MATLAB uses an inference engine to enforce language constraints for simulation and code generation. Embedded MATLAB works with Real-Time Workshop to convert code from a dynamically typed language (MATLAB) to a statically typed language (C), without using dynamic memory allocation. To convert data types accurately, the Embedded MATLAB inference engine requires that you define the class, size, and complexity of data in the source code so it can assign data types correctly at compile time.

Supported MATLAB Features

Embedded MATLAB supports the following MATLAB features:

- N-dimensional arrays
- Matrix operations
- Subscripting (see "Using Matrix Indexing Operations" on page 1-11)
- Complex numbers (see "Working with Complex Numbers" on page 1-12)
- Numeric classes (see "Supported Variable Types" on page 1-8)
- Double-precision, single-precision, and integer math

- Fixed-point arithmetic (see “Working with the Fixed-Point Embedded MATLAB™ Subset” in the Fixed-Point Toolbox documentation)
- `if`, `switch`, `while`, and `for` statements
- Subfunctions (see “Calling Functions in Embedded MATLAB” on page 1-62)
- Persistent variables (see “Declaring Persistent Variables” on page 1-10)
- Structures (see “Using Structures” on page 1-79)
- Characters (see “Working with Characters” on page 1-15)
- Function handles (see “Using Function Handles” on page 1-91)
- Frames (see “Working with Frame-Based Signals” in the Simulink documentation.
- Subset of MATLAB functions (see “Embedded MATLAB Function Library Reference” on page 1-20)
- Ability to call MATLAB functions (see “How Embedded MATLAB Resolves Function Calls” on page 1-62)

Unsupported MATLAB Features

Embedded MATLAB does not support the following MATLAB features:

- Cell arrays
- Command/function duality

Note Embedded MATLAB supports function-style syntax — but not command-style syntax — for function calls. MATLAB supports both styles (see “MATLAB Calling Syntax” in the MATLAB Programming documentation).

- Dynamic variables

Note You cannot use variables that change size.

- Global variables
- Java
- Matrix deletion
- Nested functions
- Objects
- Sparse matrices
- `Try/catch` statements

Using Embedded MATLAB with MathWorks Products

Embedded MATLAB works with the following products:

Product	Interface with Embedded MATLAB	Details
Simulink	Provides • The front-end for writing and simulating Embedded MATLAB Function blocks and Truth Table blocks in Simulink models • Embedded MATLAB MEX as the back end for generating C-MEX functions from M-code that complies with Embedded MATLAB	See: • "Using the Embedded MATLAB Function Block" in the Simulink documentation • "Building a Simulink Model with a Stateflow Truth Table" in the Stateflow documentation • Chapter 2, "Working with Embedded MATLAB MEX"
Stateflow	Provides the front-end for writing and simulating Embedded MATLAB functions in Stateflow charts.	See "Using Embedded MATLAB Functions" and "Truth Table Functions" in the Stateflow documentation.
Real-Time Workshop	Provides the back end for generating embeddable C code from Embedded MATLAB functions in Simulink and Stateflow.	See Embedded MATLAB Function in the Simulink Reference.

Product	Interface with Embedded MATLAB	Details
Fixed-Point Toolbox	Allows you to use Embedded MATLAB MEX to accelerate fixed-point algorithms in M-code.	See “Working with the Fixed-Point Embedded MATLAB™ Subset” in the Fixed-Point Toolbox documentation.
SimEvents®	Provides the front-end for writing and simulating Embedded MATLAB functions that manipulate data associated with entities.	See “Writing Functions to Manipulate Attributes”

Embedded MATLAB Coding Style

In this section...

Writing Reusable Code

With Embedded MATLAB, you can write reusable functions using standard MATLAB function file names which you can call from different locations, for example, in a Simulink model or M-function library. Embedded MATLAB can compile external functions on the MATLAB path and integrate them into generated C code for embedded targets. Embedded MATLAB can generate code for external M-functions that are compliant with Embedded MATLAB syntax and semantics. See "How Embedded MATLAB Resolves Function Calls" on page 1-62.

Common applications include:

- Overriding an Embedded MATLAB library function with a custom implementation
- Implementing a reusable library on top of standard library functions that can be used with Simulink
- Swapping between different implementations of the same function

Working with Variables

- “Initializing Persistent Variables” on page 1-10

Rules for Defining Variables

In Embedded MATLAB, you must explicitly and unambiguously define variables before using them in operations or returning them as outputs. You define variables using assignment statements. You can assign values to variables that are scalars, matrices, or structures.

Supported Variable Types

Embedded MATLAB functions support a subset of MATLAB data types represented by the following functions:

Type/Function	Description
`char`	Character array (string)
`complex`	Complex data. Cast function takes real and imaginary components
`double`	Double-precision floating point
`int8`, `int16`, `int32`	Signed integer
`logical`	Boolean `true` or `false`
`single`	Single-precision floating point
`struct`	Structure (see “Using Structures” on page 1-79)
`uint8`, `uint16`, `uint32`	Unsigned integer

Embedded MATLAB also supports fixed-point data, as described in “Working with the Fixed-Point Embedded MATLAB™ Subset” in the Fixed-Point Toolbox User’s Guide documentation.

Local Variables

In Embedded MATLAB, you define local variables implicitly in the function code, but you declare function arguments in the Model Explorer. This topic describes how to work with local variables in Embedded MATLAB and explains any exceptions or deviations from MATLAB behavior:

Creating Local Variables By Assignment. As in MATLAB, you create variables in Embedded MATLAB by assignment. Unlike MATLAB, you cannot change the size, type, or complexity of the variable after the initial assignment. Therefore, you must set these properties as part of the assignment.

For example, the following initial assignments create variables in an Embedded MATLAB function:

```
...
% a is a scalar of type double.
a = 14.7;

% b has properties of a, scalar of type double.
b = a;

% c is a 5-by-2 double array of zeros.
c = zeros(5,2);

% d has properties of c (5-by-2 double array of zeros).
d = c;

% e is 5-by-2 array of type double.
e = [1 2 3 4 5; 6 7 8 9 0];
...
```

The following rules apply when you create variables implicitly in the body of an Embedded MATLAB function:

- By default, variables are local; they do not persist between function calls. To make variables persistent, see “Declaring Persistent Variables” on page 1-10.
- Unlike in MATLAB, you cannot set the size of a variable with indexing in an assignment statement.

  For example, the following initial assignment is not allowed in Embedded MATLAB functions:

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

- You can use typecast functions in assignment statements.

  In the following example code, you declare `y` and `z` to be integers with these initial assignments:

  ```
  x = 15; % Because constants are of type double, so is x.
  y = int16(3); % y is a constant of type int16.
  z = uint8(x); % z has the value of x, but cast to uint8.
  ```

- Unlike in MATLAB, you cannot change the size, type, or complexity of variables after the initial assignment.

  In the following example, the last two statements each flag an error:

  ```
  x = 2.75 % OK
  y = [1 2; 3 4] % OK
  x = int16(x); % ERROR: cannot recast x
  y = [1 2 3; 4 5 6] %ERROR: cannot resize y
  ```

Declaring Persistent Variables

Persistent variables are local to the function in which they are declared, but they retain their values in memory between calls to the function. To declare persistent variables in your Embedded MATLAB function, use the `persistent` statement, as in this example:

```
persistent PROD_X;
```

The declaration should appear at the top of the function body, after the header and comments, but before the first use of the variable.

Initializing Persistent Variables

You initialize persistent variables in Embedded MATLAB functions the same way as in MATLAB (see Persistent Variables in the MATLAB Programming documentation). When you declare a persistent variable, Embedded MATLAB initializes its value to an empty matrix. After the declaration statement, you can assign your own value to it using the `isempty` statement, as in this example:

```
function findProduct(inputvalue)
```

```
persistent PROD_X

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

Using Matrix Indexing Operations

Embedded MATLAB supports matrix indexing operations for a matrix `M` with limitations for the following types of expressions:

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

  Embedded MATLAB never dynamically allocates memory for the size of the expressions that change as the program executes. To implement this behavior, use `for` loops as shown in the following example:

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

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

- `M(i:i+k)` where `i` is unknown but `k` is known

  In this case, since `i` — and therefore `i+k` — are not known, memory cannot be allocated for the numerical result. However, memory can be allocated for the following workaround:

  ```
  M(i + (0:k))
  ```

  In this case, an unknown scalar value `i` is added to each element of the known index vector `0...k`. This means that memory for `k+1` elements of `M` is allocated.

- Initialization of the following style:

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

  In this case, the size of M changes as the loop is executed.

Working with Complex Numbers

Embedded MATLAB supports complex numbers and operations

- "Creating Local Complex Variables By Assignment" on page 1-12
- "Semantic Rules for Complex Numbers" on page 1-14

Creating Local Complex Variables By Assignment

As in MATLAB, you create complex variables in Embedded MATLAB by assignment. Unlike MATLAB, you must set complexity at the time of assignment, either by assigning a complex constant to the variable or using the complex function, as in these examples:

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

Use the following rules to specify and use complex variables in Embedded MATLAB functions:

- Complex numbers obey the Embedded MATLAB rule that once you set the type and size of a variable, you cannot cast it to another type or size.

  In the following example, the variable x is declared complex and stays complex:

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

  Conflicts can occur from operations with real operands that can have complex results. For example, the following code generates an error:

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

The following is a possible workaround that you can use if you know that a variable can be assigned a complex number:

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

- Cases in which a function can return a complex number for a real argument are handled individually for each function.

  Generally, this can result in a complex result or a warning that the function takes only arguments producing real results. For example, for negative arguments, the function `sqrt` warns that only real positive or complex arguments are allowed.

- In general, if an expression has a complex number or variable in it, its result is a complex number, even if the result is 0.

  For example, the following code produces the complex result `z`:

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

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

  An exception to the preceding rule is a function call that takes complex arguments but produces real results, as shown in the following examples:

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

Another exception is a function call that takes real arguments but produces complex results, as shown in the following example:

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

Semantic Rules for Complex Numbers

with the following exceptions:

- The first use of a variable that is later assigned a complex result must also be complex. For example,

  ```
  X = 3;
    .
    .
    .
  X = 4 + 5i;
  ```

  fails because `X` is not defined as a complex variable by its first assignment. However,

  ```
  X = 3 + 0i;
    .
    .
    .
  X = 4 + 5i;
  ```

  succeeds because `X` is defined as a complex variable in its first assignment.

- Even if the imaginary part is zero, if the result might be complex, Embedded MATLAB will treat it as complex. For example, although

  ```
  X = ifft(fft(Y));
  ```

  yields a real answer, Embedded MATLAB assumes that the function `ifft` might return a complex result. The workaround is to use the `real` function:

  ```
  X = real(ifft(fft(Y)));
  ```

Working with Characters

Embedded MATLAB represents characters in 8 bits and, therefore, does not support the complete set of Unicode characters. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with Embedded MATLAB characters.

Supported Operators

In this section...

Control Flow Statements

Embedded MATLAB functions support the following MATLAB program statements:

Statement	Description
`break`	`break` statement
`continue`	`continue` statement
`for`	`for` statement
`if`	`if` statement The conditions of an `if` statement cannot use `&` and `|` operators. In their place, use the `&&` and `||` operators, respectively. To logically collapse vectors into scalars, use the function `all`.
`return`	`return` statement
`switch`	`switch` statement The behavior matches the MATLAB `switch` statement, which executes only the first matching case.
`while`	`while` statement The conditions of `while` statements cannot use `&` and `|` operators. In their place, use the `&&` and `||` operators, respectively. To logically collapse vectors into scalars, use the function `all`.

Arithmetic Operators

Embedded MATLAB functions support the following MATLAB arithmetic operations:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
.*	Array multiplication
/	Slash or matrix right division
./	Array right division
\	Backslash or matrix left division
.\	Array left division
^	Matrix power
.^	Array power
[ ]	Concatenation of matrices
'	Complex conjugate transpose
.'	Transpose
(r, c)	Matrix indexing, where `r` and `c` are vectors of row and column indices, respectively

See Arithmetic Operators + - * / \ ^ ' in the MATLAB Function Reference documentation for detailed descriptions of each operator.

Relational Operators

Embedded MATLAB functions support the following element-wise relational operators:

Operation	Description
<	Less than
<=	Less than or equal to
>=	Greater than or equal to
>	Greater than
==	Equal
~=	Not equal

See Relational Operators < > <= >= == ~= in the MATLAB Function Reference documentation for detailed descriptions of each operator.

Logical Operators

Embedded MATLAB functions support the following element-wise logical operators:

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>&</td><td>Logical AND

This & operator is limited to use outside if and while statement conditions. In its place, use the && operator. To logically collapse vectors into scalars, use the function all.</td></tr>
<tr><td>|</td><td>Logical OR

This | operator is limited to use outside if and while statements. In its place, use the || operator. To logically collapse vectors into scalars, use the function all.</td></tr>
<tr><td>-</td><td>Element complement</td></tr>
<tr><td>xor</td><td>Logical exclusive-OR</td></tr>
<tr><td>&&</td><td>Logical AND (short-circuiting)</td></tr>
<tr><td>||</td><td>Logical OR (short-circuiting)</td></tr>
</table>

See Logical Operators: Element-wise & | ~ and Logical Operators: Short-circuit && || in the MATLAB Function Reference documentation for detailed descriptions of each operator.

Embedded MATLAB Function Library Reference

In this section...

About Embedded MATLAB Library Functions

Each Embedded MATLAB library function has the same name, arguments, and functionality as its MATLAB, Fixed-Point Toolbox, or Signal Processing Toolbox counterpart. However, Embedded MATLAB library functions come with limitations that allow Embedded MATLAB to generate efficient embeddable code. By using this set of functions when programming in Embedded MATLAB, you can generate code for building a portable, standalone, executable target.

Note For more information on fixed-point support in Embedded MATLAB, refer to "Working with the Fixed-Point Embedded MATLAB™ Subset" in the Fixed-Point Toolbox documentation.

Embedded MATLAB Function Library — Alphabetical List

This topic lists the MATLAB functions supported by Embedded MATLAB in alphabetical order. See also "Embedded MATLAB Function Library — Categorical List" on page 1-42.

Function	Product	Remarks/Limitations
abs	MATLAB	—
abs	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
acos	MATLAB	• Generates an error during simulation and returns NaN for Real-Time Workshop targets when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
acosd	MATLAB	—
acosh	MATLAB	• Generates an error during simulation and returns NaN for Real-Time Workshop targets when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
all	MATLAB	—
all	Fixed-Point Toolbox	—
and	MATLAB	—
angle	MATLAB	—
any	MATLAB	—
any	Fixed-Point Toolbox	—
asec	MATLAB	—
asecd	MATLAB	—
asech	MATLAB	—

Function	Product	Remarks/Limitations
asin	MATLAB	• Generates an error during simulation and returns NaN for Real-Time Workshop targets when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
asind	MATLAB	—
asinh	MATLAB	—
atan	MATLAB	—
atan2	MATLAB	—
atand	MATLAB	—
atanh	MATLAB	• Generates an error during simulation and returns NaN for Real-Time Workshop targets when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
beta	MATLAB	—
betainc	MATLAB	—
betaln	MATLAB	—
bitand	MATLAB	• Does not support floating-point inputs. The arguments must belong to an integer class.
bitand	Fixed-Point Toolbox	—
bitandreduce	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
bitcmp	MATLAB	• Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitcmp	Fixed-Point Toolbox	—
bitconcat	Fixed-Point Toolbox	—
bitget	MATLAB	—
bitget	Fixed-Point Toolbox	—
bitor	MATLAB	• Does not support floating-point inputs. The arguments must belong to an integer class.
bitor	Fixed-Point Toolbox	—
bitorreduce	Fixed-Point Toolbox	—
bitrevorder	Signal Processing Toolbox	• Requires Signal Processing Blockset license.
bitrol	Fixed-Point Toolbox	—
bitror	Fixed-Point Toolbox	—
bitset	MATLAB	• Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitset	Fixed-Point Toolbox	—
bitshift	MATLAB	• Does not support floating-point input for the first argument. The first argument must belong to an integer class.

Function	Product	Remarks/Limitations
`bitshift`	Fixed-Point Toolbox	—
`bitsliceget`	Fixed-Point Toolbox	—
`bitsll`	Fixed-Point Toolbox	
`bitsra`	Fixed-Point Toolbox	—
`bitsrl`	Fixed-Point Toolbox	—
`bitxor`	MATLAB	• Does not support floating-point inputs. The arguments must belong to an integer class.
`bitxor`	Fixed-Point Toolbox	—
`bitxorreduce`	Fixed-Point Toolbox	—
`bsxfun`	MATLAB	—
`cart2pol`	MATLAB	—
`cart2sph`	MATLAB	—
`cast`	MATLAB	—
`cat`	MATLAB	• Accepts up to three input arguments.
`ceil`	MATLAB	—
`char`	MATLAB	—
`chol`	MATLAB	• Does not allow two output arguments.
`class`	MATLAB	—
`compan`	MATLAB	—
`complex`	MATLAB	—
`complex`	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Toolbox	—
conv	MATLAB	—
conv2	MATLAB	—
cos	MATLAB	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	—
coth	MATLAB	—
cov	MATLAB	—
cross	MATLAB	• If supplied, dim must be a constant.
csc	MATLAB	—
cscd	MATLAB	—
csch	MATLAB	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Toolbox	—
cumprod	MATLAB	—
cumsum	MATLAB	—
cumtrapz	MATLAB	—
deconv	MATLAB	—
det	MATLAB	—

Function	Product	Remarks/Limitations
`detrend`	MATLAB	• If supplied and not empty, the input argument `bp` must satisfy the following requirements: - Be real - Be sorted in ascending order - Restrict elements to integers in the interval `[1, n-2]`, where `n` is the number of elements in a column of input argument `X` , or the number of elements in `X` when `X` is a row vector - Contain all unique values
`diag`	MATLAB	• If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
`diag`	Fixed-Point Toolbox	—
`diff`	MATLAB	• If supplied, the arguments representing the number of times to apply `diff` and the dimension along which to calculate the difference must be constants.
`disp`	Fixed-Point Toolbox	—
`divide`	Fixed-Point Toolbox	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object. • Complex and imaginary divisors are not supported.
`dot`	MATLAB	—
`double`	MATLAB	—
`double`	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
`eig`	MATLAB	• `QZ` algorithm used in all cases. Consequently, for the standard eigenvalue problem (`B` identity), results will be similar to those obtained using the following in MATLAB: `[V,D] = eig(A,eye(size(A)),'qz')` However, `V` may represent a different basis of eigenvectors, and the eigenvalues in `D` may not be in the same order. • Options `'balance'`, `'nobalance'`, and `'chol'` are not supported. • Outputs are always of complex type.
`ellipke`	MATLAB	—
`end`	Fixed-Point Toolbox	—
`eps`	MATLAB	—
`eps`	Fixed-Point Toolbox	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix single and double signals.
`eq`	MATLAB	—
`eq`	Fixed-Point Toolbox	• Not supported for fixed-point signals with different biases.
`erf`	MATLAB	—
`erfc`	MATLAB	—
`erfcinv`	MATLAB	—
`erfcx`	MATLAB	—
`erfinv`	MATLAB	—
`exp`	MATLAB	—
`expint`	MATLAB	—

Function	Product	Remarks/Limitations
`expm`	MATLAB	—
`expm1`	MATLAB	—
`eye`	MATLAB	• Dimensions must be real, nonnegative, integer constants.
`factorial`	MATLAB	—
`false`	MATLAB	• Dimensions must be real, nonnegative, integer constants.
`fft`	MATLAB	• Length of input vector must be a power of 2.
`fftshift`	MATLAB	—
`fi`	Fixed-Point Toolbox	• Use to create a fixed-point constant or variable in Embedded MATLAB. • The default constructor syntax without any input arguments is not supported. • The syntax `fi('PropertyName',PropertyValue...)` is not supported. To use property name/property value pairs, you must first specify the value `v` of the `fi` object as in `fi(v,'PropertyName',PropertyValue...)`. • Works for constant input values only; that is, the value of the input must be known at compile time. • `numerictype` object information must be available for non-fixed-point Simulink inputs.
`filter`	MATLAB	—
`filter2`	MATLAB	—

Function	Product	Remarks/Limitations
fimath	Fixed-Point Toolbox	• Fixed-point signals coming in to an Embedded MATLAB Function block from Simulink are assigned the fimath object defined in the Embedded MATLAB Function dialog in the Model Explorer. • Use to create fimath objects in Embedded MATLAB code.
fix	MATLAB	—
flipdim	MATLAB	—
fliplr	MATLAB	—
flipud	MATLAB	—
floor	MATLAB	—
freqspace	MATLAB	—
gamma	MATLAB	—
gammainc	MATLAB	—
gammaln	MATLAB	—
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Toolbox	• Not supported for fixed-point signals with different biases.
get	Fixed-Point Toolbox	• The syntax structure = get(o) is not supported.
getlsb	Fixed-Point Toolbox	—
getmsb	Fixed-Point Toolbox	—
gt	MATLAB	—
gt	Fixed-Point Toolbox	• Not supported for fixed-point signals with different biases.

Function	Product	Remarks/Limitations
`hilb`	MATLAB	—
`histc`	MATLAB	—
`horzcat`	Fixed-Point Toolbox	—
`hypot`	MATLAB	—
`idivide`	MATLAB	• `opt` string must be in lowercase. • For efficient generated code, MATLAB divide-by-zero rules are supported only for the `'round'` option.
`ifft`	MATLAB	• Length of input vector must be a power of 2. • Output of `ifft` block is always complex.
`ifftshift`	MATLAB	—
`imag`	MATLAB	—
`imag`	Fixed-Point Toolbox	—
`ind2sub`	MATLAB	• No support for N-dimensional matrices. Size vector must have exactly two elements.
`inf`	MATLAB	• Dimensions must be real, nonnegative, integer constants.
`int8`, `int16`, `int32`	MATLAB	—
`int8`, `int16`, `int32`	Fixed-Point Toolbox	—
`interp1`	MATLAB	• Supports only `linear` and `nearest` interpolation methods. • Does not handle evenly spaced `X` indices separately. • `X` must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.

Function	Product	Remarks/Limitations
`interp1q`, see `interp1`	MATLAB	• `X` must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.
`intmax`	MATLAB	—
`intmin`	MATLAB	—
`inv`	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB.
`invhilb`	MATLAB	—
`ipermute`	MATLAB	—
`isa`	MATLAB	—
`ischar`	MATLAB	—
`iscolumn`	Fixed-Point Toolbox	—
`isempty`	MATLAB	—
`isempty`	Fixed-Point Toolbox	—
`isequal`	MATLAB	• Supports only two arguments.
`isfi`	Fixed-Point Toolbox	—
`isfimath`	Fixed-Point Toolbox	—
`isfinite`	MATLAB	—
`isfinite`	Fixed-Point Toolbox	—
`isfloat`	MATLAB	—
`isinf`	MATLAB	—
`isinf`	Fixed-Point Toolbox	—
`isinteger`	MATLAB	—

Function	Product	Remarks/Limitations
islogical	MATLAB	—
isnan	MATLAB	—
isnan	Fixed-Point Toolbox	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Toolbox	—
isnumerictype	Fixed-Point Toolbox	—
isreal	MATLAB	—
isreal	Fixed-Point Toolbox	—
isrow	Fixed-Point Toolbox	—
isscalar	MATLAB	—
isscalar	Fixed-Point Toolbox	—
issigned	Fixed-Point Toolbox	—
issorted	MATLAB	—
isstruct	MATLAB	—
isvector	MATLAB	—
isvector	Fixed-Point Toolbox	—
kron	MATLAB	
lcm	MATLAB	—
ldivide	MATLAB	—
le	MATLAB	—

Function	Product	Remarks/Limitations
`le`	Fixed-Point Toolbox	• Not supported for fixed-point signals with different biases.
`length`	MATLAB	—
`length`	Fixed-Point Toolbox	—
`linspace`	MATLAB	• Number of points *N* must be a constant that is positive, real, and integer valued
`log`	MATLAB	• Generates an error during simulation and returns `NaN` for Real-Time Workshop targets when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
`log2`	MATLAB	—
`log10`	MATLAB	—
`log1p`	MATLAB	—
`logical`	MATLAB	—
`logical`	Fixed-Point Toolbox	—
`logspace`	MATLAB	—
`lowerbound`	Fixed-Point Toolbox	—
`lsb`	Fixed-Point Toolbox	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix single and double signals.
`lt`	MATLAB	—
`lt`	Fixed-Point Toolbox	• Not supported for fixed-point signals with different biases.
`lu`	MATLAB	—
`magic`	MATLAB	—

Function	Product	Remarks/Limitations
`max`	MATLAB	—
`max`	Fixed-Point Toolbox	—
`mean`	MATLAB	—
`median`	MATLAB	—
`meshgrid`	MATLAB	—
`min`	MATLAB	—
`min`	Fixed-Point Toolbox	—
`minus`	MATLAB	—
`minus`	Fixed-Point Toolbox	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
`mldivide`	MATLAB	—
`mod`	MATLAB	—
`mode`	MATLAB	• Does not support third output argument `C` (cell array)
`mpower`	MATLAB	—
`mrdivide`	MATLAB	—
`mtimes`	MATLAB	—
`mtimes`	Fixed-Point Toolbox	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.

Function	Product	Remarks/Limitations
NaN or nan	MATLAB	• Dimensions must be real, nonnegative, integer constants • Supports only one or two dimension arguments
nargin	MATLAB	—
nargout	MATLAB	—
nchoosek	MATLAB	—
ndims	MATLAB	—
ndims	Fixed-Point Toolbox	—
ne	MATLAB	—
ne	Fixed-Point Toolbox	• Not supported for fixed-point signals with different biases.
nextpow2	MATLAB	—
norm	MATLAB	—
normest	MATLAB	—
not	MATLAB	—
nthroot	MATLAB	—
numberofelements	Fixed-Point Toolbox	• numberofelements and numel both work the same as MATLAB numel for fi objects in Embedded MATLAB.
numerictype	Fixed-Point Toolbox	• Fixed-point signals coming in to an Embedded MATLAB Function block from Simulink are assigned a numerictype object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a nonfixed-point signal. • Use to create numerictype objects in Embedded MATLAB code.

Function	Product	Remarks/Limitations
`ones`	MATLAB	• Dimensions must be real, non-negative, integer constants
`or`	MATLAB	—
`pascal`	MATLAB	—
`permute`	MATLAB	—
`permute`	Fixed-Point Toolbox	—
`pi`	MATLAB	—
`pinv`	MATLAB	—
`planerot`	MATLAB	—
`plus`	MATLAB	—
`plus`	Fixed-Point Toolbox	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
`pol2cart`	MATLAB	—
`poly`	MATLAB	• Does not discard non-finite input values • Complex input always produces complex output
`polyfit`	MATLAB	—
`polyval`	MATLAB	—
`pow2`	Fixed-Point Toolbox	• For the syntax `pow2(a, K)`, `K` must be a constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.

Function	Product	Remarks/Limitations
power	MATLAB	• Generates an error during simulation and returns NaN for Real-Time Workshop targets when both X and Y are real, but power(X,Y) is complex. To get the complex result, make the input value X complex by passing in complex(X). For example, power(complex(X),Y). • Generates an error during simulation and returns NaN for Real-Time Workshop targets when both X and Y are real, but X .^ Y is complex. To get the complex result, make the input value X complex by using complex(X). For example, complex(X).^Y.
prod	MATLAB	—
qr	MATLAB	—
rand	MATLAB	• Does not support the V5 generator. • May not match MATLAB if seeded with negative values.
randn	MATLAB	• May not match MATLAB if seeded with negative values
range	Fixed-Point Toolbox	—
rank	MATLAB	—
rcond	MATLAB	—
rdivide	MATLAB	—
real	MATLAB	—
real	Fixed-Point Toolbox	—
reallog	MATLAB	—
realmax	MATLAB	—
realmax	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
`realmin`	MATLAB	—
`realmin`	Fixed-Point Toolbox	—
`realpow`	MATLAB	—
`realsqrt`	MATLAB	—
`rem`	MATLAB	—
`repmat`	MATLAB	—
`repmat`	Fixed-Point Toolbox	—
`rescale`	Fixed-Point Toolbox	—
`reshape`	MATLAB	• Accepts a maximum of three arguments
`reshape`	Fixed-Point Toolbox	—
`rot90`	MATLAB	—
`round`	MATLAB	—
`sec`	MATLAB	—
`secd`	MATLAB	—
`sech`	MATLAB	—
`shiftdim`	MATLAB	• Second argument must be a constant • Class of second argument must be `single` or `double`
`sign`	MATLAB	—
`sign`	Fixed-Point Toolbox	—
`sin`	MATLAB	—
`sind`	MATLAB	—
`single`	MATLAB	—

Function	Product	Remarks/Limitations
`single`	Fixed-Point Toolbox	—
`sinh`	MATLAB	—
`size`	MATLAB	—
`size`	Fixed-Point Toolbox	—
`sort`	MATLAB	—
`sortrows`	MATLAB	—
`sosfilt`	Signal Processing Toolbox	• Requires Signal Processing Blockset license
`sph2cart`	MATLAB	—
`squeeze`	MATLAB	—
`sqrt`	MATLAB	• Generates an error during simulation and returns `NaN` for Real-Time Workshop targets when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
`sqrt`	Fixed-Point Toolbox	• Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
`std`	MATLAB	—
`strcmp`	MATLAB	• Arguments must be computable at compile time.
`struct`	MATLAB	—
`sub2ind`	MATLAB	• Does not support N-dimensional matrices. Size vector must have exactly two elements. • Maximum number of input arguments is three.
`subsasgn`	Fixed-Point Toolbox	—
`subspace`	MATLAB	—

Function	Product	Remarks/Limitations
subsref	Fixed-Point Toolbox	—
sum	MATLAB	—
sum	Fixed-Point Toolbox	—
svd	MATLAB	—
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—
times	MATLAB	—
times	Fixed-Point Toolbox	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
toeplitz	MATLAB	—
trace	MATLAB	—
trapz	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Toolbox	—
tril	MATLAB	—
tril	Fixed-Point Toolbox	—
triu	MATLAB	—
triu	Fixed-Point Toolbox	—
true	MATLAB	• Dimensions must be real, non-negative, integer constants

Function	Product	Remarks/Limitations
`typecast`	MATLAB	• Value of string input argument `type` must be lower case • Data type of input argument `X` can be inherited in Embedded MATLAB Function blocks only if `class(X)` is `'double'`; otherwise, you must specify input port data types explicitly
`uint8`, `uint16`, `uint32`	MATLAB	—
`uint8`, `uint16`, `uint32`	Fixed-Point Toolbox	—
`uminus`	MATLAB	—
`uminus`	Fixed-Point Toolbox	—
`uplus`	MATLAB	—
`uplus`	Fixed-Point Toolbox	—
`upperbound`	Fixed-Point Toolbox	—
`vander`	MATLAB	—
`var`	MATLAB	—
`vertcat`	Fixed-Point Toolbox	—
`wilkinson`	MATLAB	—
`xcorr`	Signal Processing Toolbox	• Does not support the case where A is a matrix • Does not support partial (abbreviated) strings of biased, unbiased, coeff, or none • Requires Signal Processing Blockset license
`xor`	MATLAB	—
`zeros`	MATLAB	• Dimensions must be real, non-negative, integer constants

Embedded MATLAB Function Library — Categorical List

The following topics list functions in the Embedded MATLAB library by different function types. Each entry includes a function name link to online help for the equivalent MATLAB or Fixed-Point Toolbox function along with a one-line description.

For an alphabetical list of these functions, and remarks and limitations for them, see “Embedded MATLAB Function Library — Alphabetical List” on page 1-20.

Arithmetic Operator Functions

See Arithmetic Operators + - * / \ ^ ’ in the MATLAB Function Reference documentation for detailed descriptions of the following operator equivalent functions.

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

Casting Functions

Embedded MATLAB functions support the following functions for converting one type of data to another:

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

Complex Number Functions

Embedded MATLAB functions support the following functions for complex numbers:

Function	Description
complex	Construct complex data from real and imaginary components
conj	Return the conjugate of a complex number
imag	Return the imaginary part of a complex number
isnumeric	True for numeric arrays
isreal	Return false (0) for a complex number
isscalar	True if array is a scalar
real	Return the real part of a complex number

Derivative and Integral Functions

Embedded MATLAB functions support the following functions for derivatives and integrals:

Function	Description
`cumtrapz`	Cumulative trapezoidal numerical integration
`diff`	Differences and approximate derivatives
`trapz`	Trapezoidal numerical integration

Discrete Math Functions

Embedded MATLAB functions support the following discrete math functions:

Function	Description
`lcm`	Least common multiple of corresponding elements in arrays
`gcd`	Return an array containing the greatest common divisors of the corresponding elements of integer arrays
`nchoosek`	Binomial coefficient or all combinations

Exponential Functions

Embedded MATLAB functions support the following exponential functions:

Function	Description
`exp`	Exponential
`expm`	Matrix exponential
`expm1`	Compute `exp(x)-1` accurately for small values of x
`factorial`	Factorial function
`log`	Natural logarithm
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`log10`	Common (base 10) logarithm

Function	Description
`log1p`	Compute `log(1+x)` accurately for small values of `x`
`nextpow2`	Next higher power of 2
`nthroot`	Real nth root of real numbers
`reallog`	Natural logarithm for nonnegative real arrays
`realpow`	Array power for real-only output
`realsqrt`	Square root for nonnegative real arrays
`sqrt`	Square root

Filtering and Convolution Functions

Embedded MATLAB functions support the following filtering and convolution functions:

Function	Description
`conv`	Convolution and polynomial multiplication
`conv2`	2-D convolution
`deconv`	Deconvolution and polynomial division
`detrend`	Remove linear trends
`filter`	1-D digital filter
`filter2`	2-D digital filter

Fixed-Point Toolbox Functions

For more information on fixed-point support in Embedded MATLAB, see "Working with the Fixed-Point Embedded MATLAB™ Subset" in the Fixed-Point Toolbox documentation. Embedded MATLAB supports the following functions from Fixed-Point Toolbox:

Function	Description
`abs`	Absolute value of `fi` object
`all`	Determine whether all array elements are nonzero

Function	Description
any	Determine whether any array elements are nonzero
bitand	Bitwise AND of two fi objects
bitandreduce	Bitwise AND of consecutive range of bits
bitcmp	Bitwise complement of fi object
bitconcat	Concatenate bits of two fi objects
bitget	Bit at certain position
bitor	Bitwise OR of two fi objects
bitorreduce	Bitwise OR of consecutive range of bits
bitrol	Bitwise rotate left
bitror	Bitwise rotate right
bitset	Set bit at certain position
bitshift	Shift bits specified number of places
bitsliceget	Consecutive slice of bits
bitsll	Bit shift left logical
bitsra	Bit shift right arithmetic
bitsrl	Bit shift right logical
bitxor	Bitwise exclusive OR of two fi objects
bitxorreduce	Bitwise exclusive OR of consecutive range of bits
complex	Construct complex fi object from real and imaginary parts
conj	Complex conjugate of fi object
ctranspose	Complex conjugate transpose of fi object
diag	Diagonal matrices or diagonals of matrix
disp	Display object
divide	Divide two objects
double	Double-precision floating-point real-world value of fi object
end	Last index of array

Function	Description
`eps`	Quantized relative accuracy for `fi` or `quantizer` objects
`eq`	Determine whether real-world values of two `fi` objects are equal
`fi`	Construct `fi` object
`fimath`	Construct `fimath` object
`ge`	Determine whether real-world value of one `fi` object is greater than or equal to another
`get`	Property values of object
`getlsb`	Least significant bit
`getmsb`	Most significant bit
`gt`	Determine whether real-world value of one `fi` object is greater than another
`horzcat`	Horizontally concatenate multiple `fi` objects
`imag`	Imaginary part of complex number
`int8`, `int16`, or `int32`	Stored integer value of `fi` object as built-in `int8`, `int16`, or `int32`
`iscolumn`	Determine whether `fi` object is column vector
`isempty`	Determine whether array is empty
`isfi`	Determine whether variable is `fi` object
`isfimath`	Determine whether variable is `fimath` object
`isfinite`	Determine whether array elements are finite
`isinf`	Determine whether array elements are infinite
`isnan`	Determine whether array elements are NaN
`isnumeric`	Determine whether input is numeric array
`isnumerictype`	Determine whether variable is `numerictype` object
`isreal`	Determine whether array elements are real
`isrow`	Determine whether `fi` object is row vector
`isscalar`	Determine whether input is scalar

Function	Description
issigned	Determine whether fi object is signed
isvector	Determine whether input is vector
le	Determine whether real-world value of fi object is less than or equal to another
length	Vector length
logical	Convert numeric values to logical
lowerbound	Lower bound of range of fi object
lsb	Scaling of least significant bit of fi object
lt	Determine whether real-world value of one fi object is less than another
max	Largest element in array of fi objects
min	Smallest element in array of fi objects
minus	Matrix difference between fi objects
mtimes	Matrix product of fi objects
ndims	Number of array dimensions
ne	Determine whether real-world values of two fi objects are not equal
numberofelements	Number of data elements in fi array
numerictype	Construct numerictype object
permute	Rearrange dimensions of multidimensional array
plus	Matrix sum of fi objects
pow2	Multiply by 2^K
range	Numerical range of fi or quantizer object
real	Real part of complex number
realmax	Largest positive fixed-point value or quantized number
realmin	Smallest positive normalized fixed-point value or quantized number
repmat	Replicate and tile array
rescale	Change scaling of fi object

Function	Description
`reshape`	Reshape array
`sign`	Perform `signum` function on array
`single`	Single-precision floating-point real-world value of `fi` object
`size`	Array dimensions
`sqrt`	Square root of `fi` object
`subsasgn`	Subscripted assignment
`subsref`	Subscripted reference
`sum`	Sum of array elements
`times`	Element-by-element multiplication of `fi` objects
`transpose`	Transpose operation
`tril`	Lower triangular part of matrix
`triu`	Upper triangular part of matrix
`uint8`, `uint16`, or `uint32`	Stored integer value of `fi` object as built-in `uint8`, `uint16`, or `uint32`
`uminus`	Negate elements of `fi` object array
`uplus`	Unary plus
`upperbound`	Upper bound of range of `fi` object
`vertcat`	Vertically concatenate multiple `fi` objects

Histogram Functions

Embedded MATLAB functions support the following histogram functions:

Function	Description
`histc`	Histogram count

Input and Output Functions

Embedded MATLAB functions support the following functions for accessing argument and return values:

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

Interpolation and Computational Geometry

Embedded MATLAB functions support the following functions for interpolation and computational geometry:

Function	Description
cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
interp1	One-dimensional interpolation (table lookup)
interp1q	Quick one-dimensional linear interpolation (table lookup)
meshgrid	Generate X and Y arrays for 3-D plots
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Logical Operator Functions

Embedded MATLAB functions support the following functions for performing logical operations:

Function	Description
and	Logical AND (&)
bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places

Function	Description
bitxor	Bitwise XOR
not	Logical NOT (~)
or	Logical OR (|)
xor	Logical exclusive-OR

Matrix and Array Functions

Embedded MATLAB functions support the following functions for matrices and arrays:

Function	Description
abs	Return absolute value and complex magnitude of an array
all	Test if all elements are nonzero
angle	Phase angle
any	Test for any nonzero elements
bsxfun	Applies element-by-element binary operation to two arrays with singleton expansion enabled
cat	Concatenate arrays along specified dimension
compan	Companion matrix
cond	Condition number of a matrix with respect to inversion
cov	Covariance matrix
cross	Vector cross product
cumprod	Cumulative product of array elements
cumsum	Cumulative sum of array elements
det	Matrix determinant
diag	Return a matrix formed around the specified diagonal vector and the specified diagonal (0, 1, 2,...) it occupies
diff	Differences and approximate derivatives
dot	Vector dot product

Function	Description
eig	Eigenvalues and eigenvectors
eye	Identity matrix
false	Return an array of 0s for the specified dimensions
flipdim	Flip array along specified dimension
fliplr	Flip matrix left to right
flipud	Flip matrix up to down
hilb	Hilbert matrix
ind2sub	Subscripts from linear index
isequal	Test arrays for equality
isvector	Determine whether input is vector
inv	Inverse of a square matrix
invhilb	Inverse of Hilbert matrix
ipermute	Inverse permute dimensions of array
isempty	Determine whether array is empty
isfinite	Detect finite elements of an array
isfloat	Determine if input is floating-point array
isinf	Detect infinite elements of an array
isinteger	Determine if input is integer array
islogical	Determine if input is logical array
isnan	Detect NaN elements of an array
kron	Kronecker tensor product
length	Return the length of a matrix
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
lu	Matrix factorization
magic	Magic square

Function	Description
max	Maximum elements of a matrix
min	Minimum elements of a matrix
ndims	Number of dimensions
norm	Vector and matrix norms
normest	2-norm estimate
ones	Create a matrix of all 1s
pascal	Pascal matrix
permute	Rearrange dimensions of array
pinv	Pseudoinverse of a matrix
planerot	Givens plane rotation
prod	Product of array element
qr	Orthogonal-triangular decomposition
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
repmat	Replicate and tile an array
reshape	Reshape one array into the dimensions of another
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sign	Signum function
size	Return the size of a matrix
sort	Sort elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
sub2ind	Single index from subscripts
subspace	Angle between two subspaces
sum	Sum of matrix elements

Function	Description
toeplitz	Toeplitz matrix
trace	Sum of diagonal elements
tril	Extract lower triangular part
triu	Extract upper triangular part
true	Return an array of logical (Boolean) 1s for the specified dimensions
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix
zeros	Create a matrix of all zeros

Polynomial Functions

Embedded MATLAB functions support the following functions for polynomials:

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

Relational Operator Functions

Embedded MATLAB functions support the following functions for performing relational operations:

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

Rounding and Remainder Functions

Embedded MATLAB functions support the following rounding and remainder functions:

Function	Description
ceil	Round toward plus infinity
fix	Round toward zero
floor	Round toward minus infinity
mod	Modulus (signed remainder after division)
rem	Remainder after division
round	Round toward nearest integer

Set Functions

Embedded MATLAB functions support the following set functions:

Function	Description
issorted	Determine whether set elements are in sorted order

Signal Processing Functions

Embedded MATLAB supports the following signal processing functions:

Function	Description
`bitrevorder`	Permute data into bit-reversed order (requires Signal Processing Blockset license)
`chol`	Cholesky factorization
`conv`	Convolution and polynomial multiplication
`fft`	Discrete Fourier transform
`fftshift`	Shift zero-frequency component to center of spectrum
`filter`	Filter a data sequence using a digital filter that works for both real and complex inputs
`freqspace`	Frequency spacing for frequency response
`ifft`	Inverse discrete Fourier transform
`ifftshift`	Inverse discrete Fourier transform shift
`sosfilt`	Second-order (biquadratic) IIR filtering (requires Signal Processing Blockset license)
`svd`	Singular value decomposition
`xcorr`	Cross-correlation function estimates (requires Signal Processing Blockset license)

Special Values

Embedded MATLAB functions support the following special data values:

Symbol	Description
`eps`	Floating-point relative accuracy
`inf`	IEEE arithmetic representation for positive infinity
`intmax`	Largest possible value of specified integer type
`intmin`	Smallest possible value of specified integer type
NaN or `nan`	Not a number
`pi`	Ratio of the circumference to the diameter for a circle
`rand`	Uniformly distributed pseudorandom numbers

Symbol	Description
randn	Normally distributed random numbers
realmax	Largest positive floating-point number
realmin	Smallest positive floating-point number

Specialized Math

Embedded MATLAB functions support the following specialized math functions:

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

Statistical Functions

Embedded MATLAB functions support the following statistical functions:

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

String Functions

Embedded MATLAB functions support the following functions for handling strings:

Function	Description
`char`	Create character array (string)
`ischar`	True for character array (string)
`strcmp`	Return a logical result for the comparison of two strings; limited to strings known at compile time

Structure Functions

Embedded MATLAB functions support the following functions for handling structures:

Function	Description
`struct`	Create structure
`isstruct`	Determine whether input is a structure

Trigonometric Functions

Embedded MATLAB functions support the following trigonometric functions:

Function	Description
acos	Inverse cosine
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse cosecant and inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atan2	Four quadrant inverse tangent
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine
cosd	Cosine; result in degrees
cosh	Hyperbolic cosine
cot	Cotangent; result in radians
cotd	Cotangent; result in degrees
coth	Hyperbolic cotangent
csc	Cosecant; result in radians
cscd	Cosecant; result in degrees

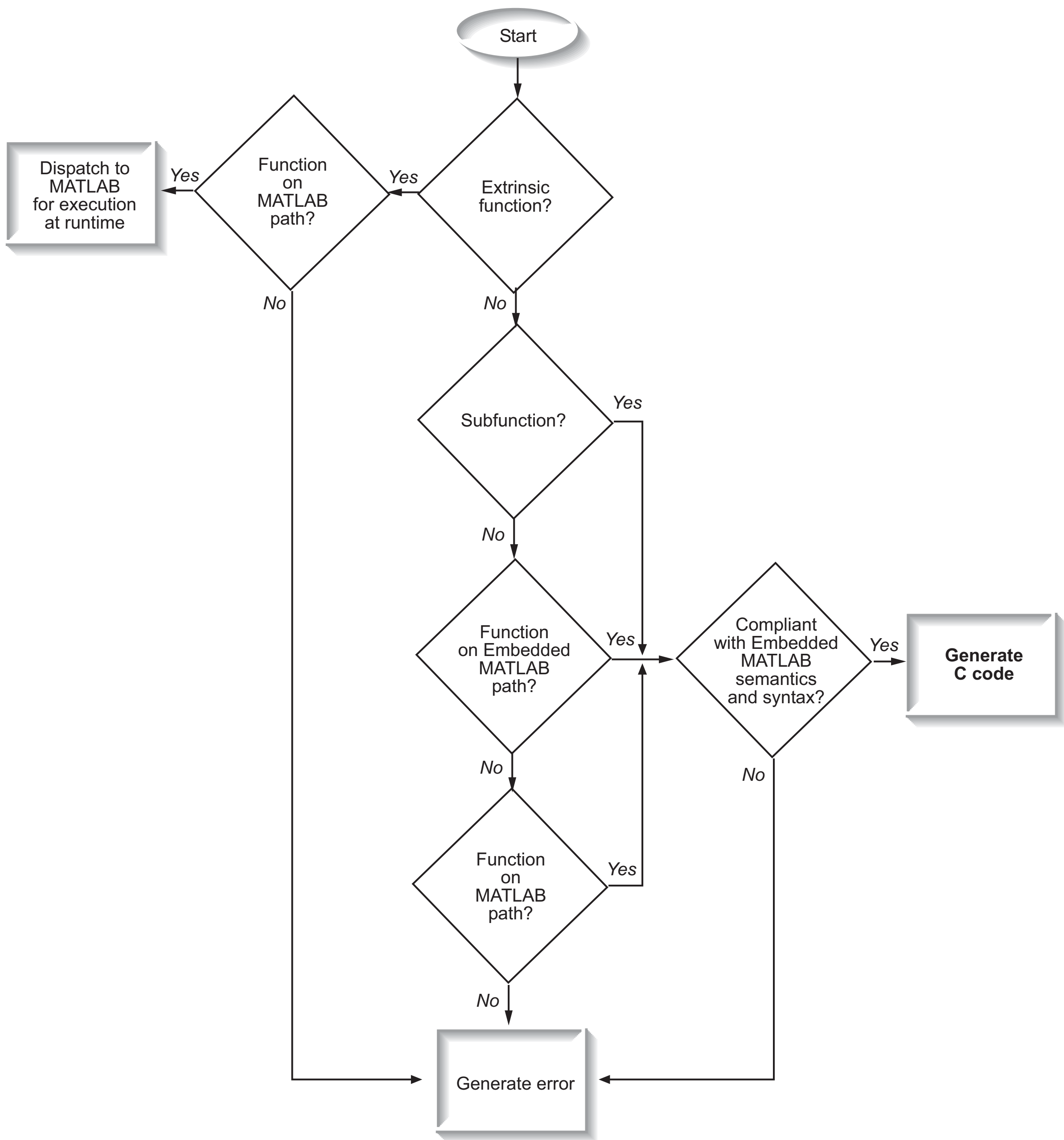
Function	Description
`csch`	Hyperbolic cosecant
`hypot`	Square root of sum of squares
`sec`	Secant; result in radians
`secd`	Secant; result in degrees
`sech`	Hyperbolic secant
`sin`	Sine
`sind`	Sine; result in degrees
`sinh`	Hyperbolic sine
`tan`	Tangent
`tand`	Tangent; result in degrees
`tanh`	Hyperbolic tangent

Calling Functions in Embedded MATLAB

In this section...

How Embedded MATLAB Resolves Function Calls

From an Embedded MATLAB function, you can call subfunctions, Embedded MATLAB library functions, and external MATLAB functions. Embedded MATLAB resolves function names as follows:


Start
Extrinsic function?
Yes
Function on MATLAB path?
Yes
Dispatch to MATLAB for execution at runtime
No
No
Subfunction?
Yes
No
Function on Embedded MATLAB path?
Yes
Compliant with Embedded MATLAB semantics and syntax?
Yes
Generate C code
No
No
Function on MATLAB path?
Yes
No
Generate error

Key Points About Resolving Function Calls

The diagram illustrates key points about how Embedded MATLAB resolves function calls:

- Embedded MATLAB searches two paths, the Embedded MATLAB path and the MATLAB path.

  See "Compile Path Search Order" on page 1-64.

- Embedded MATLAB attempts to compile all functions unless you explicitly declare them to be extrinsic.

  An extrinsic function is an M-function on the MATLAB path that Embedded MATLAB dispatches to MATLAB for execution. Embedded MATLAB does not generate code for extrinsic functions. You declare functions to be extrinsic by using the Embedded MATLAB function `eml.extrinsic`, as described in "Declaring MATLAB Functions as Extrinsic Functions" on page 1-71.

After Embedded MATLAB resolves a function name, it then resolves the file type based on precedence rules described in "How Embedded MATLAB Resolves File Types on the Path" on page 1-65.

Compile Path Search Order

Embedded MATLAB searches two paths to resolve function calls:

1. Embedded MATLAB path

   Embedded MATLAB searches this path first. The Embedded MATLAB path contains the Embedded MATLAB library functions.

2. MATLAB path

   If Embedded MATLAB does not find the function on the Embedded MATLAB path, it searches the MATLAB path.

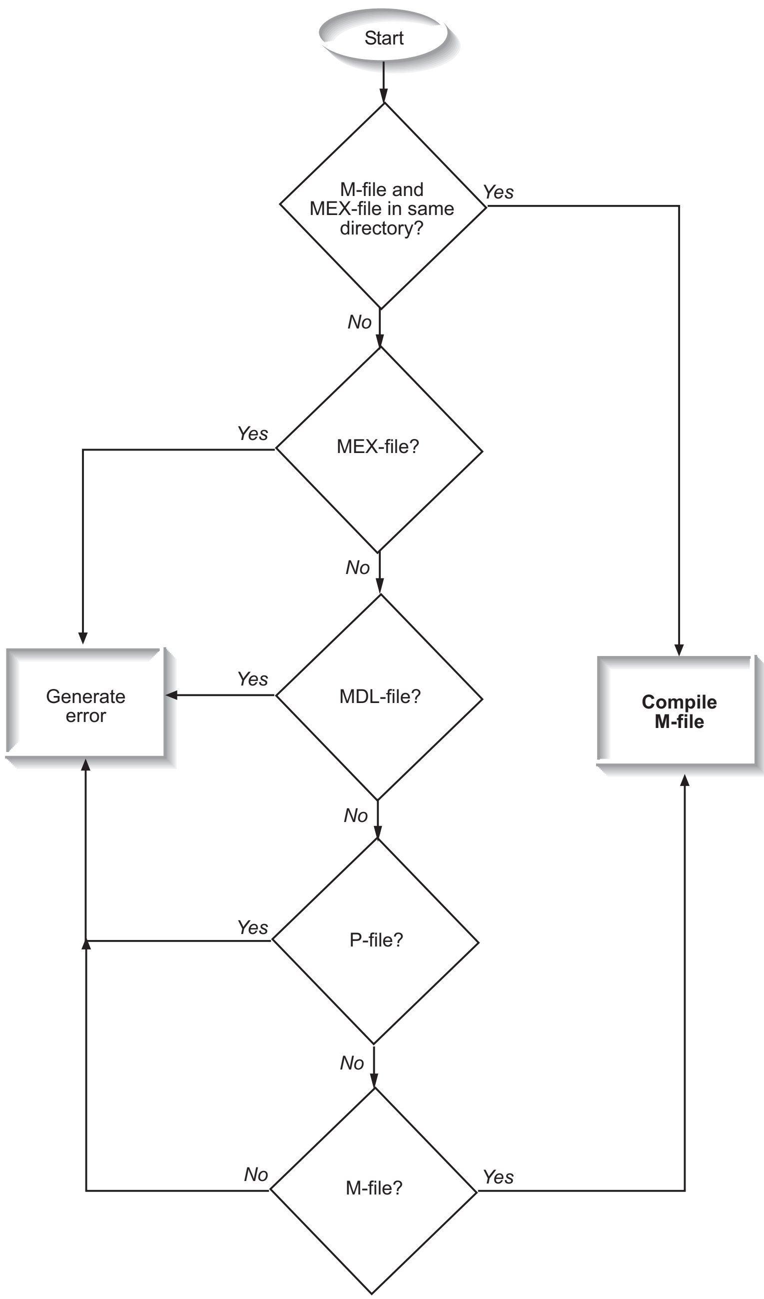
Embedded MATLAB applies MATLAB dispatcher rules when searching each path. For more information, see "How MATLAB Determines Which Method to Call" in the MATLAB Programming documentation.

When to Use the Embedded MATLAB Path

Use the Embedded MATLAB path to override a MATLAB function with a customized version. Since Embedded MATLAB searches the Embedded MATLAB path first, an M-file on the Embedded MATLAB path always shadows an M-file of the same name on the MATLAB path.

How Embedded MATLAB Resolves File Types on the Path

After Embedded MATLAB resolves function names on a path, it resolves file types on the path using precedence rules as follows:


Start
M-file and
MEX-file in same
directory?
Yes
No
MEX-file?
Yes
No
MDL-file?
Yes
No
Generate
error
Compile
M-file
P-file?
Yes
No
M-file?
No
Yes

Adding the Compilation Directive %#eml

To instruct Embedded MATLAB to compile an external function, add the `%#eml` pragma to the function code. Adding this pragma accomplishes two purposes:

- If compilation succeeds, provides a visual cue that the function complies with the Embedded MATLAB subset
- If compilation fails, produces detailed diagnostic messages to help you correct violations of Embedded MATLAB syntax and semantics

If compilation fails and your function code does not contain the `%#eml` pragma, Embedded MATLAB prompts you to clarify whether you really want to compile the external function or whether you want to dispatch it to MATLAB for execution, as follows:

To:	Do This:
Compile with detailed diagnostics	Add `%#eml` to the function code and recompile.
Dispatch the function to MATLAB	Declare the function to be extrinsic and recompile. See “Declaring MATLAB Functions as Extrinsic Functions” on page 1-71, and recompile.

A Simple Example

Suppose you have an Embedded MATLAB Function block that contains the function `example` which, in turn, calls the function `foo` on the MATLAB path. Here is the code for `example.m`:

```
function example(u)

foo(u);
```

Here is the code for `foo.m`:

```
function y = foo(u)
y = 0;
for i = 1:u:u
    y = y+i;
```

```
end
```

When you build `example.m`, Embedded MATLAB tries to compile `foo.m` based on the heuristic described in "How Embedded MATLAB Resolves Function Calls" on page 1-62. During compilation, Embedded MATLAB detects errors and generates the following messages:

- Warning:

  ```
  Please add an %#eml pragma to 'D:/Work/foo.m'
  to indicate that this file is intended to be
  used for Embedded MATLAB.
  ```

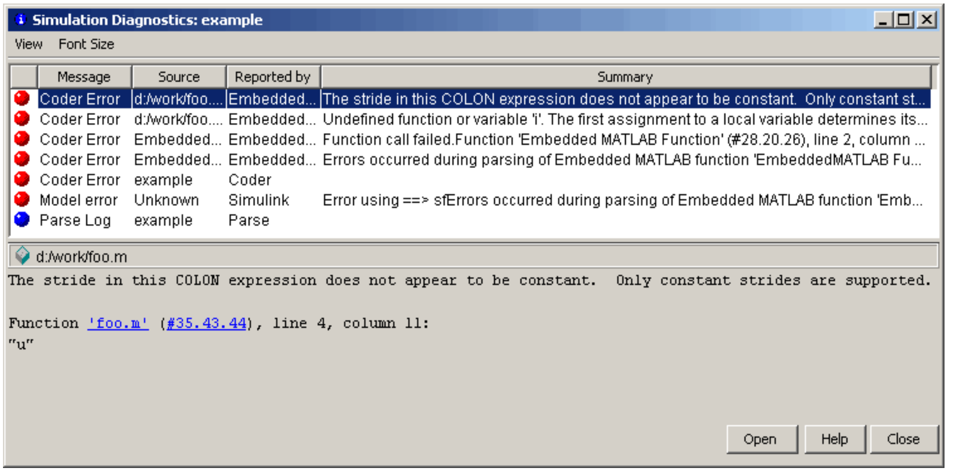
- Error:

  ```
  The function failed to compile, if you intended
  to compile the file use %#eml directive in
  'D:/Work/foo.m', however if you wish to call out
  to matlab use eml.extrinsic('foo') before calling
  the function.
  ```

To continue with compilation, add `%#eml` to `foo.m`, shown highlighted below:

```
%#eml
function y = foo(u)
y = 0;
for i = 1:u:u
    y = y+i;
end
```

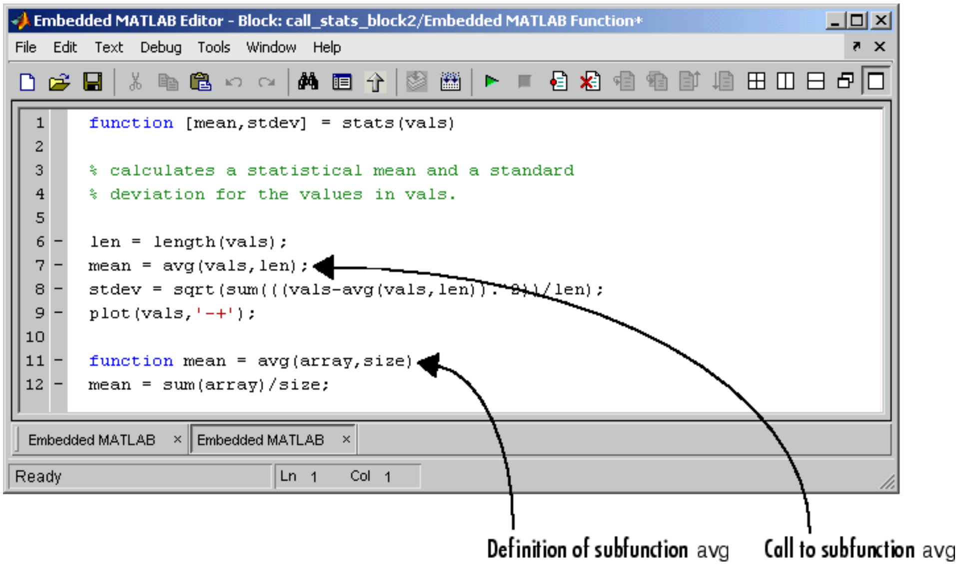
When you build `example.m` again, you get a more detailed message, explaining the specific syntax violations, as follows:



Calling Subfunctions

Subfunctions are functions defined in the body of an Embedded MATLAB function. They work the same way in Embedded MATLAB functions as they do in MATLAB.

The following example illustrates how to define and call a subfunction in an Embedded MATLAB function:



You can include subfunctions for Embedded MATLAB functions just as you would in MATLAB M-file functions. Subfunctions can have multiple arguments and return values, using any types and sizes supported by Embedded MATLAB. See "Subfunctions" in the MATLAB Programming documentation for more information.

Calling Embedded MATLAB Library Functions

You can call Embedded MATLAB library functions directly. The Embedded MATLAB function library is a subset of MATLAB, Fixed-Point Toolbox, and Signal Processing Toolbox functions which can be used to generate code. For a list of supported functions appear in "Embedded MATLAB Function Library Reference" on page 1-20.

For more information about fixed-point support in Embedded MATLAB, refer to "Working with the Fixed-Point Embedded MATLAB™ Subset" in the Fixed-Point Toolbox documentation.

Calling MATLAB Functions

Embedded MATLAB attempts to compile all MATLAB functions unless you explicitly declare them to be extrinsic (see "How Embedded MATLAB Resolves Function Calls" on page 1-62). An extrinsic function is a function that is executed by MATLAB during simulation. Embedded MATLAB does not compile or generate code for extrinsic functions (see "How Embedded MATLAB Resolves Extrinsic Functions" on page 1-74).

There are two methods for declaring a function extrinsic in Embedded MATLAB:

- Declare the function extrinsic in Embedded MATLAB main functions or subfunctions (see "Declaring MATLAB Functions as Extrinsic Functions" on page 1-71).
- Call the MATLAB function indirectly using `feval` (see "Calling MATLAB Functions Using feval" on page 1-73).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function extrinsic, add a declaration at the top of the main Embedded MATLAB function or a subfunction using this syntax:

```
eml.extrinsic('function_name_1', ... , 'function_name_n');
```

For example, the following code declares the MATLAB `find` function extrinsic in the main Embedded MATLAB function `foo`:

```
function y = foo

eml.extrinsic('find');

x = ones(4);
y = x;
y = find(x);
```

When to Use the eml.extrinsic Declaration. Use the `eml.extrinsic` declaration to:

- Call MATLAB functions that produce no output — such as `plot` — for visualizing results during simulation, without generating unnecessary code (see “How Embedded MATLAB Resolves Extrinsic Functions” on page 1-74).

- Make your code self-documenting and easier to debug. You can scan the source code for `eml.extrinsic` declarations to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with mxArrays” on page 1-75).
- Save typing. With one declaration, you ensure that each subsequent function call is extrinsic, as long as the call and the declaration are in the same scope (see “Scope of Extrinsic Function Declarations” on page 1-72).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 1-72). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using feval” on page 1-73).

Rules for Extrinsic Function Declarations. Observe the following rules when declaring functions extrinsic in Embedded MATLAB:

- You must declare the function extrinsic before you call it.
- You cannot use the extrinsic declaration in conditional statements.

Scope of Extrinsic Function Declarations. The `eml.extrinsic` declaration has function scope. For example, consider the following code:

```
function y = foo
eml.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, Embedded MATLAB interprets the functions `rat` and `min` as extrinsic every time they are called in the main function `foo`.

There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a subfunction, as in this example:

  ```
  function y = foo
  ```

```
eml.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = mymin(N, D);

function y = mymin(a,b)
eml.extrinsic('min');
y = min(a,b);
```

Here, the function `rat` is extrinsic every time it is called inside the main function `foo`, but the function `min` is extrinsic only when called inside the subfunction `mymin`.

- Call the MATLAB function using `feval`, as described in "Calling MATLAB Functions Using feval" on page 1-73.

Calling MATLAB Functions Using feval

Embedded MATLAB automatically interprets the function `feval` as an extrinsic function. Therefore, you can use `feval` to conveniently call MATLAB functions from Embedded MATLAB.

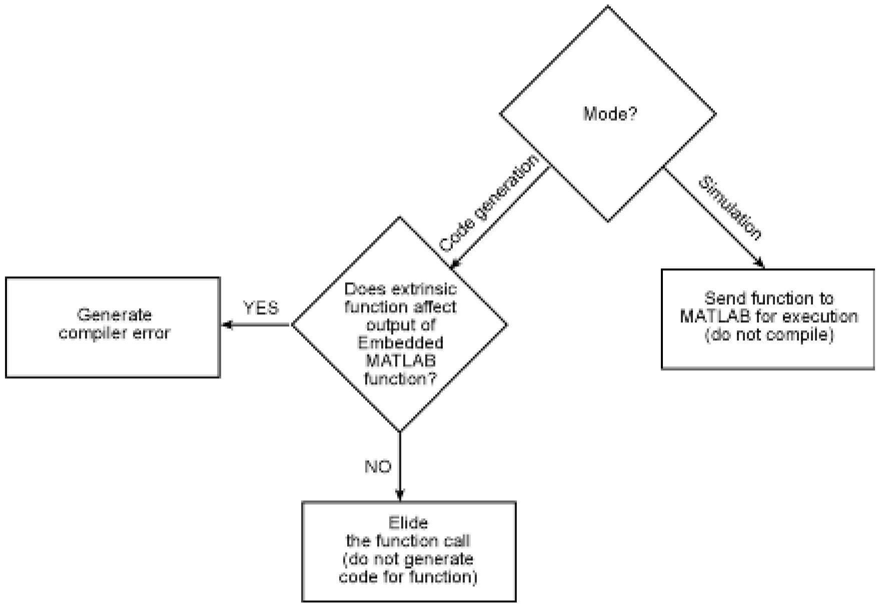
Consider the following example:

```
function y = foo
eml.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because `feval` is extrinsic, the statement `feval('min', N, D)` is evaluated by MATLAB — not Embedded MATLAB — which has the same effect as declaring the function `min` extrinsic for just this one call. By contrast, the function `rat` is extrinsic throughout the function `foo`.

How Embedded MATLAB Resolves Extrinsic Functions

Embedded MATLAB resolves extrinsic functions as follows:



For simulation targets, Embedded MATLAB generates code for the call to a MATLAB function, but does not generate the function's internal code. Embedded MATLAB sends the extrinsic function to MATLAB for execution. Therefore, you can run the simulation only on platforms where MATLAB is installed.

For Real-Time Workshop® and custom targets, Embedded MATLAB attempts to determine whether the extrinsic function affects the output of the Embedded MATLAB function in which it is called — for example by returning `mxArrays` to an output variable (see "Working with mxArrays" on page 1-75). If Embedded MATLAB can determine that there is no effect on output, Embedded MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, Embedded MATLAB issues a compiler error.

Working with mxArrays

The output of an extrinsic function is an `mxArray` — also called MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables
- Passing `mxArrays` to functions and returning them from functions
- Converting `mxArrays` to known types at run time

To use `mxArrays` returned by extrinsic functions in other operations, you must first convert them to known types, as described in "Converting mxArrays to Known Types" on page 1-75.

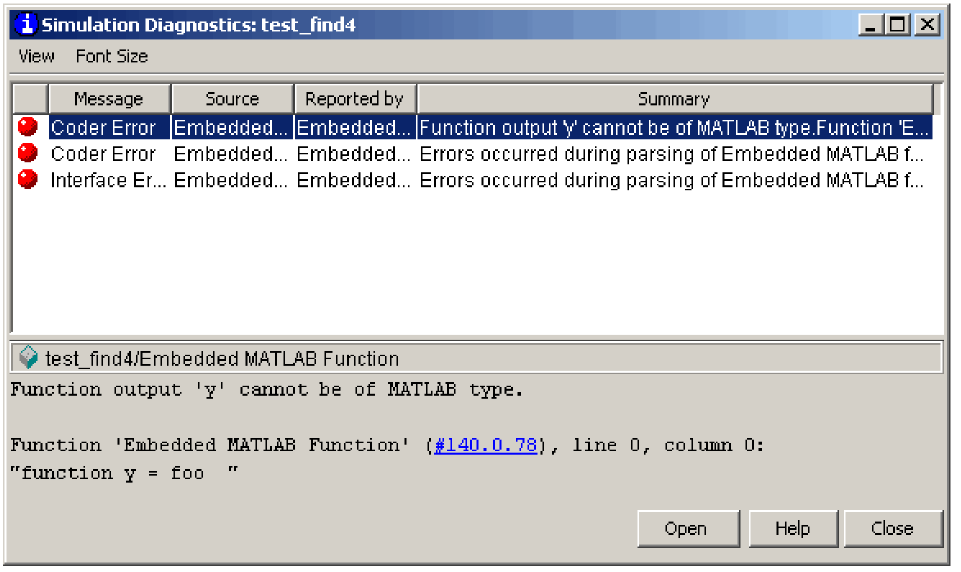
Converting mxArrays to Known Types. To convert an`mxArray` to a known type, assign the `mxArray` to a variable whose type is defined. At run time, Embedded MATLAB converts the `mxArray` to the type of the variable assigned to it. However, if the data in the `mxArray` is not consistent with the type of the variable, Embedded MATLAB generates an error.

For example, consider this code:

```
function y = foo
eml.extrinsic('rat','min');
[N D] = rat(pi);
y = min(N, D);
```

Here, the top-level Embedded MATLAB function `foo` calls the extrinsic MATLAB function `rat`, which returns two `mxArrays` representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these `mxArrays` to another extrinsic MATLAB function — in this case, `min` — you cannot assign the `mxArray` returned by `min` to the output `y`.

If you run this function `foo` in an Embedded MATLAB Function block in Simulink, the code generates the following error during simulation:



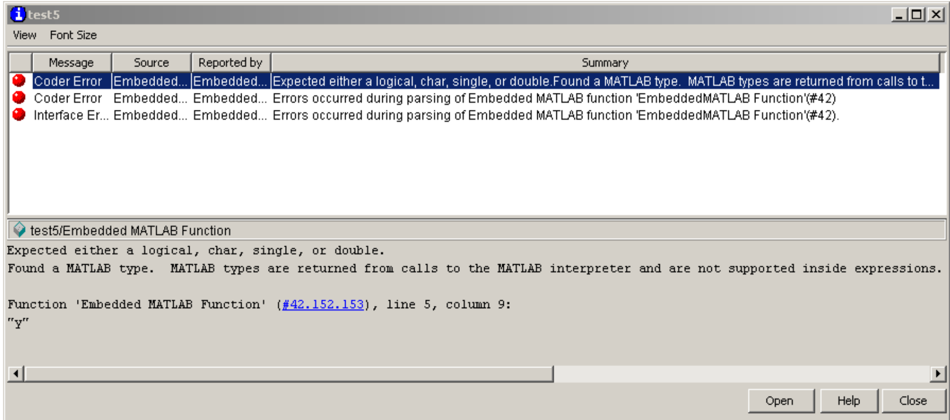
To correct this problem, declare `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
function y = foo
eml.extrinsic('rat','min');
[N D] = rat(pi);
y = 0; % y is a scalar of type double
y = min(N,D);
```

In the next example, Embedded MATLAB attempts to use an `mxArray` in an arithmetic expression:

```
function z = foo
eml.extrinsic('find');
x = ones(1); % x is a 1-by-1 array of type double
y = find(x); % y is a 1-by-1 array of type mxArray
z = x + y;
```

If you run this function `foo` in an Embedded MATLAB Function block in Simulink, the code generates a compiler error during simulation because it attempts to add the `mxArray` `y` to a double array `x`:



The value `y` is an `mxArray` because the code assigns it the `mxArray` value returned by the extrinsic MATLAB function `find`. To prevent this error, you must declare `y` to be the same type and size as `x` — a 1-by-1 matrix of type `double` — before assigning `y` to the return value of `find(x)`, as in this example:

```
function z = foo
eml.extrinsic('find');
x = ones(1); % x is a 1-by-1 array of type double
y = ones(1); % y is a 1-by-1 array of type double
y = find(x); % y returned from find converted to
             % 1-by-1 array of type double
z = x + y;
```

Here, the Embedded MATLAB function `ones(1)` returns a 1-by-1 matrix of type `double`, thereby converting `y` to the same type and size as `x` at run time. Now that `y` is defined, Embedded MATLAB can convert the `mxArray` returned by `find(x)` to a known type — an array of type `double` — at run time for assignment to `y`. As a result, the expression `z = x + y` adds variables of the same type and does not generate an error.

Restrictions on Extrinsic Functions in Embedded MATLAB

As a subset of MATLAB, Embedded MATLAB does not support the full MATLAB run-time environment. Therefore, Embedded MATLAB imposes the following restrictions when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller or write to the caller's workspace do not work when the caller is an Embedded MATLAB function, including:
  - `dbstack`
  - `evalin`
  - `assignin`
- The MATLAB debugger cannot inspect variables in Embedded MATLAB functions.
- Embedded MATLAB may produce unpredictable results if your extrinsic function performs any of the following actions at run time:
  - Change directories
  - Change the MATLAB path
  - Delete or add M-files
  - Change warning states
  - Change MATLAB preferences
  - Change Simulink parameters

Limit on Function Arguments

Embedded MATLAB allows you to call functions with up to 64 inputs and 64 outputs.

Using Structures

In this section...

About Embedded MATLAB Structures

The Embedded MATLAB structure is a data type that is based on the MATLAB structure (see "Structures" in the MATLAB Programming documentation). By imposing some restrictions, Embedded MATLAB compiles MATLAB structures to generate efficient C code in Real-Time Workshop®. Structures in Embedded MATLAB support a subset of the operations available for MATLAB structures. In Embedded MATLAB, you can:

- Define structures as local or persistent variables inside Embedded MATLAB functions
- Define primary function inputs as structures
- Pass structures to subfunctions
- Index structure fields using dot notation

Elements of Embedded MATLAB Structures

The elements of Embedded MATLAB structures are called fields. Like structures in MATLAB, the fields of an Embedded MATLAB structure can contain data of any type and size, including:

- Scalars
- Strings
- Composite data, such as other structures
- Arrays of structures

Note Unlike structure arrays in MATLAB, each structure field in an Embedded MATLAB array must have the same type, size, and complexity (see "Limitations with Structures" on page 1-86).

Types of Embedded MATLAB Structures

You can define the following types of Embedded MATLAB structures:

Type	How to Define	Details
Input	Depends on how you use Embedded MATLAB: • Define structures in Embedded MATLAB Function blocks based on Simulink bus objects (requires Simulink) • Define primary function inputs based on structure definitions in the MATLAB workspace (to generate C-MEX code using Embedded MATLAB MEX, as described in Chapter 2, "Working with Embedded MATLAB MEX")	See: • "Using Bus Objects" in the Simulink User's Guide documentation • "Specifying Properties of Primary Function Inputs" on page 2-13
Output	Define structure variable in Embedded MATLAB function	See "Defining Outputs as Structures" on page 1-84.

Type	How to Define	Details
Local	Define local structure variable in Embedded MATLAB function	See “Defining Local Structure Variables” on page 1-81.
Persistent	Declare structure variable to be persistent in Embedded MATLAB function	See “Making Structures Persistent” on page 1-85.

Defining Local Structure Variables

You can define local structures as variables inside Embedded MATLAB functions. Local structures are temporary by default, but you can make them persistent (see “Making Structures Persistent” on page 1-85).

You can define structures explicitly as scalars or arrays, as described in these topics:

- “Defining Scalar Structures” on page 1-81
- “Defining Arrays of Structures” on page 1-83

Defining Scalar Structures

There are several ways to create scalar structures in Embedded MATLAB:

- “Defining Scalar Structures by Extension” on page 1-81
- “Defining Scalar Structures Using the MATLAB struct Function” on page 1-82
- “Defining Scalar Structures by Assignment” on page 1-82

Defining Scalar Structures by Extension. You can create scalar structures by extension by adding fields to a variable using dot notation. For example, the following code creates a structure to represent a point `p` with coordinates `x`, `y`, and `z`:

```
...
p.x = 1;
p.y = 3;
```

```
p.z = 1;
...
```

You can also nest scalar structures in direct assignment statements by appending more than one field to a variable using dot notation. For example, the following code adds a color field to structure p:

```
...
p.color.red = .2;
p.color.green = .4;
p.color.blue = .7;
...
```

See "Indexing SubStructures and Fields" on page 1-85.

Defining Scalar Structures Using the MATLAB struct Function. You can create scalar structures in Embedded MATLAB using the MATLAB `struct` function (see "Structures" in the MATLAB Programming documentation). When using `struct` in Embedded MATLAB functions, the field arguments must be scalar values. You cannot create structures of cell arrays in Embedded MATLAB. However, you can define arrays of structures, as described in "Defining Arrays of Structures" on page 1-83.

Defining Scalar Structures by Assignment. You can define scalar structures by assigning them to preexisting structures. In the following example, p is defined as a structure that has the same properties as the predefined structure S:

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

Note You do not need to predefine the variable to which you assign the structure — in this case, p. However, if you have already defined the variable, it must have the same class, size, and complexity as the structure you assign to it.

Defining Arrays of Structures

When you create an array of structures in Embedded MATLAB, you must be sure that each structure field in the array has the same size, type, and complexity (see "Limitations with Structures" on page 1-86). There are several ways to create arrays of structures in Embedded MATLAB:

- "Defining an Array of Structures from a Scalar Structure" on page 1-83
- "Defining an Array of Structures Using Concatenation" on page 1-84

Defining an Array of Structures from a Scalar Structure. You can create an array of structures from a scalar structure by using the MATLAB `repmat` function, which replicates and tiles an existing scalar structure. Follow these steps:

1. Create a scalar structure, as described in "Defining Scalar Structures" on page 1-81.

2. Call `repmat`, passing the scalar structure and the dimensions of the array.

3. Assign values to each structure using standard array indexing and structure dot notation.

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

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,3);
X(1).a = 1;
X(2).a = 2;
X(3).a = 3;
X(1).b = 4;
X(2).b = 5;
X(3).b = 6;
...
```

Defining an Array of Structures Using Concatenation. To create a small array of structures, you can use the concatenation operator, square brackets ([ ]), to join one or more structures into an array (see "Concatenating Matrices" in the MATLAB Programming documentation). In Embedded MATLAB, all the structures that you concatenate must have the same size, class, and complexity.

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

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

function s = sab(a,b)
  s.a = a;
  s.b = b;
...
```

Defining Outputs as Structures

You define primary function outputs as structures the same way you define local structures (see "Defining Local Structure Variables" on page 1-81). For example, the following code defines output `y` as a scalar structure by extension:

```
function y = fcn(u)
y.a = 1;
y.b = 2;
...
```

The next example defines output `y` as a structure with the same fields and values as in the previous example, but this time using the MATLAB `struct` function:

```
function y = fcn(u)
y = struct('a',1,'b',2);
...
```

You can also define outputs as structures by assigning them to a preexisting structure, as in this example:

```
function y = fcn(u)
x = struct('a',1,'b',2);
```

```
y = x;
...
```

See “Structures” in the MATLAB Programming documentation.

Making Structures Persistent

To make structures persist, you declare them to be persistent variables and initialize them with the `isempty` statement, as described in “Declaring Persistent Variables” on page 1-10.

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

```
function f(u)
persistent X

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

Indexing SubStructures and Fields

As in MATLAB, you index substructures and fields of Embedded MATLAB structures by using dot notation. Unlike MATLAB, you must reference field values individually (see “Reference Field Values Individually from Structure Arrays” on page 1-89).

For example, the following code excerpt from an Embedded MATLAB function uses dot notation to index fields and substructures:

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

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

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

```
...
```

The following table shows how Embedded MATLAB resolves symbols in dot notation for indexing elements of the structures in this example:

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

Assigning Values to Structures and Fields

You can assign values to any Embedded MATLAB structure, substructure, or field. Here are the guidelines:

Operation	Conditions
Assign one structure to another structure.	You must define each structure with the same number, type, and size of fields (see "Using Structures" on page 1-79).
Assign one structure to a substructure of a different structure and vice versa.	You must define the structure with the same number, type, and size of fields as the substructure.
Assign an element of one structure to an element of another structure.	The elements must have the same type and size.

Limitations with Structures

Embedded MATLAB supports MATLAB structures with the following limitations to allow efficient code generation in C:

- "Add Fields in Consistent Order" on page 1-87
- "Do Not Assign Empty Matrices" on page 1-88

- "Do Not Assign mxArrays to Structures" on page 1-88
- "Do Not Add New Fields After First Use of Structures" on page 1-88
- "Make Structures Uniform in Arrays" on page 1-89
- "Do Not Reference Fields Dynamically" on page 1-89
- "Do Not Use Field Values as Constants" on page 1-89
- "Reference Field Values Individually from Structure Arrays" on page 1-89

Add Fields in Consistent Order

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

```
function y = fcn(u)
if u > 0
   x.a = 10;
   x.b = 20;
else
   x.b = 30;  % Generates an error (on variable x)
   x.a = 40;
end
y = x.a + x.b;
```

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

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

Do Not Assign Empty Matrices

You cannot assign empty matrices to structure fields.

Do Not Assign mxArrays to Structures

You cannot assign `mxArrays` to structure elements in Embedded MATLAB; you must first convert them to known types (see "Working with mxArrays" on page 1-75).

Do Not Add New Fields After First Use of Structures

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

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

For example, consider this code:

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

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

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

```
function y = fcn(u)

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

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

Make Structures Uniform in Arrays

Each structure field in an array of structures must have the same size, type, and complexity.

Do Not Reference Fields Dynamically

You cannot reference fields in a structure by using dynamic names, which express the field as a variable expression that MATLAB evaluates at run time (see "Using Dynamic Field Names" in the MATLAB Programming documentation).

Do Not Use Field Values as Constants

Embedded MATLAB never considers the values stored in the fields of a structure to be constant values. Therefore, you cannot use field values to set the size or class of other data. For example, the following code generates an error:

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

In this example, even though you set field `a` of structure `Y` to the value 3, Embedded MATLAB does not consider `Y.a` to be a constant and, therefore, it is not a valid argument to pass to the function `zeros`.

Reference Field Values Individually from Structure Arrays

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

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

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

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

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See "Defining Local Structure Variables" on page 1-81 for more information.

Using Function Handles

In this section...

About Function Handles

You can use function handles in Embedded MATLAB to invoke functions indirectly and parameterize operations that you repeat frequently (see "Function Handles" in the MATLAB Programming documentation). In Embedded MATLAB, you can perform the following operations with function handles:

- Define handles that reference user-defined functions and built-in functions supported by Embedded MATLAB (see "Embedded MATLAB Function Library Reference" on page 1-20)

  Note You cannot define handles that reference extrinsic MATLAB functions (see "Calling MATLAB Functions" on page 1-71).

- Define function handles as scalar values
- Pass function handles as arguments to other functions (excluding extrinsic functions)

Embedded MATLAB does not support the full set the operations you can perform with function handles in MATLAB, as described in "Limitations with Function Handles" on page 1-94

Example: Defining and Passing Function Handles in an Embedded MATLAB Function

The following code example defines and calls function handles in an Embedded MATLAB function. You can copy it as is to an Embedded MATLAB Function block in Simulink or Embedded MATLAB function in Stateflow. To convert this function to a C-MEX function using `emlmex`, uncomment the two calls to the `assert` function, highlighted below:

```
function addval(m)


  % Declare class and size of primary input m
  % Uncomment next two lines to build C-MEX function with emlmex
  % assert(isa(m,'double'));
  % assert(all (size(m) == [3 3]));

  % Declare MATLAB function disp to be extrinsic
  eml.extrinsic('disp');

  disp(m);

  % Pass function handle to addone
  %  to add one to each element of m
  m = map(@addone, m);
  disp(m);

  % Pass function handle to addtwo
  %  to add two to each element of m
  m = map(@addtwo, m);
  disp(m);

  function y = map(f,m)
    y = m;
    for i = 1:numel(y)
       y(i) = f(y(i));
    end

  function y = addone(u)
  y = u + 1;
```

```
function y = addtwo(u)
y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

If you have Simulink or Fixed-Point Toolbox, you can use Embedded MATLAB MEX to convert this M-function `addval` to a C-MEX executable that you can run in MATLAB. Follow these steps:

1 At the MATLAB command prompt, issue this command:

```
emlmex addval
```

Embedded MATLAB MEX checks your code for compliance with Embedded MATLAB.

2 Define and initialize a 3-by-3 matrix by typing a command like this at the MATLAB prompt:

```
m = zeros(3)
```

3 Execute the function by typing this command:

```
addvals(m)
```

You should see the following result:

```
0   0   0
0   0   0
0   0   0

1   1   1
1   1   1
1   1   1

3   3   3
3   3   3
3   3   3
```

For more information about Embedded MATLAB MEX, see Chapter 2, "Working with Embedded MATLAB MEX".

Limitations with Function Handles

Embedded MATLAB supports MATLAB function handles with the following limitations:

- Function handles must be scalar values. You cannot store function handles in matrices or structures.
- After you bind a variable to a specific function, you cannot use the same variable to reference two different function handles, as in this example

  ```
  %Incorrect code
  ...
  x = @plus;
  x = @minus;
  ...
  ```

  This code produces a compilation error in Embedded MATLAB.
- You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions.
- You cannot pass function handles as input to primary functions in Embedded MATLAB.

  For example, consider this Embedded MATLAB function:

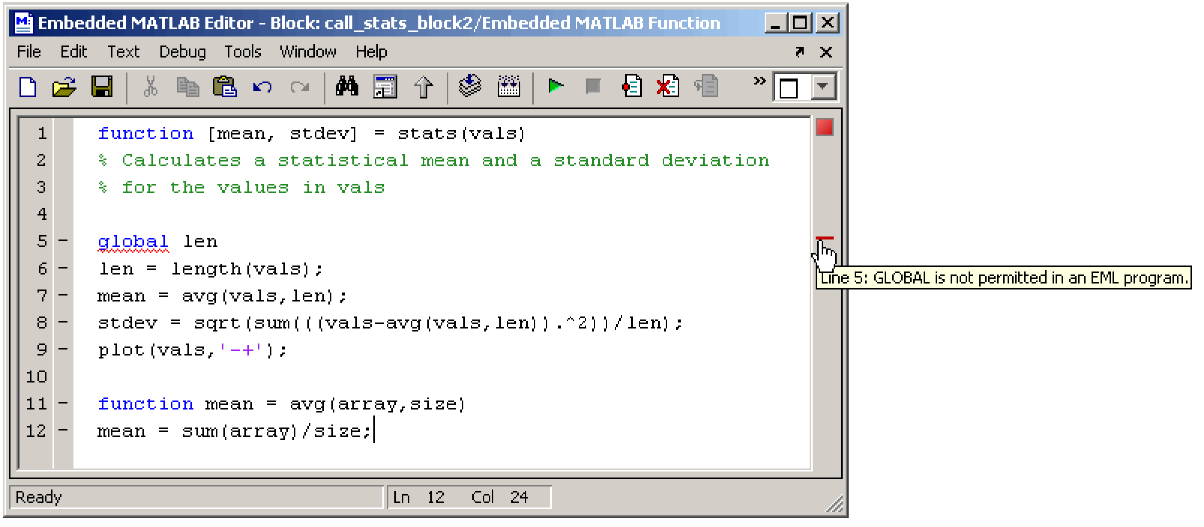
  ```
  function x = plotFcn(fhandle, data)

  assert(isa(fhandle,'function_handle') && isa(data,'double'));

  eml.extrinsic('plot');

  plot(data, fhandle(data));
  x = fhandle(data);
  ```

  In this example, the function `plotFcn` receives a function handle and its data as primary inputs. `plotFcn` attempts to call the function referenced by the `fhandle` with the input `data` and plot the results. However, you will not be able to compile `plotFcn` with Embedded MATLAB MEX. Instead, you get an error, indicating that the function `isa` does not recognize `'function_handle'` as a class name when called inside an Embedded MATLAB function to specify properties of primary inputs.

- You cannot display or watch function handles from the Embedded MATLAB debugger; they appear as empty matrices.

Using M-Lint with Embedded MATLAB

The Embedded MATLAB Editor uses the MATLAB M-Lint Code Analyzer to automatically check your Embedded MATLAB function code for errors and recommend corrections. The editor displays the same type of M-Lint bar that appears in the MATLAB editor to highlight offending lines of code. However, in the Embedded MATLAB Editor, the M-Lint bar displays Embedded MATLAB diagnostics as well as MATLAB messages, as in the following example:



For information about how to use M-Lint, see “M-Lint Code Analyzer” in the MATLAB Desktop Tools and Development Environment documentation.

Working with Embedded MATLAB MEX

About Embedded MATLAB MEX

In this section...

Optimizes M-Code for Run-Time Efficiency

Embedded MATLAB MEX is a function that optimizes M-code for run-time efficiency in the following applications:

Application	What Embedded MATLAB MEX Does
Accelerate fixed-point calculations	Converts M-code to C-MEX functions that contain Embedded MATLAB optimizations for automatically accelerating fixed-point algorithms to compiled C code speed.
Run M-code in Simulink	Checks M-code for compliance with the syntax and semantics of Embedded MATLAB, as described in Chapter 1, "Working with Embedded MATLAB". You can add compliant code to: • Embedded MATLAB Function blocks and Truth Table blocks in Simulink • Embedded MATLAB functions and Truth Table functions in Stateflow

MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute. C-MEX files are MEX-files that are written in the C programming language, but can run directly in MATLAB. For more information, see "Introducing MEX-Files" in the MATLAB External Interfaces documentation.

Running a Demo for Embedded MATLAB MEX

Fixed-Point Toolbox ships with a demonstration of how to generate a C-MEX function from M-code. The M-code takes the weighted average of a signal to create a lowpass filter. If you have a license for Fixed-Point Toolbox, you can run the demo by following these steps:

1. Install Fixed-Point Toolbox.

   Note For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

   ---

2. At the MATLAB prompt, type this command:

   ```
   demos
   ```

   The Help browser appears, listing categories of demos in the left pane.

3. In the left pane, navigate to **Toolboxes > Fixed-Point > Fixed-Point Lowpass Filtering Using Embedded MATLAB MEX**.

4. Follow the instructions in the right pane of the Help browser.

How Embedded MATLAB MEX Resolves Function Calls

Embedded MATLAB MEX resolves function calls by first searching the Embedded MATLAB path and then the MATLAB path. By default, Embedded MATLAB MEX tries to compile and generate code for functions it finds on the path unless you explicitly declare the function to be extrinsic. Embedded MATLAB does not compile extrinsic functions, but rather dispatches them to MATLAB for execution. For more information, see “How Embedded MATLAB Resolves Function Calls” on page 1-62 in the Embedded MATLAB documentation.

Workflow for Converting M-Code to a C-MEX Function

Follow these steps to convert M-code to a C-MEX function that complies with Embedded MATLAB, as described in Chapter 1, "Working with Embedded MATLAB":

Step	Action	Details
1	Install prerequisite products.	See "Installing Embedded MATLAB MEX" on page 2-5.
2	Set up your C compiler.	See "Setting Up the C Compiler" on page 2-6.
3	Set up your file infrastructure.	See "Setting Up File Infrastructure and Paths" on page 2-7.
4	Make your M-code compliant with Embedded MATLAB.	See "Making M-Code Compliant with Embedded MATLAB " on page 2-10.
5	Specify properties of primary function inputs.	See "Specifying Properties of Primary Function Inputs" on page 2-13.
6	Run Embedded MATLAB MEX with the appropriate command-line options.	See "Compiling Your M-File" on page 2-29.

Installing Embedded MATLAB MEX

Embedded MATLAB MEX ships with Simulink and Fixed-Point Toolbox. To use Embedded MATLAB MEX, you must install the following products:

- MATLAB
- Simulink and/or Fixed-Point Toolbox
- C compiler

For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

For instructions on installing and setting up a C compiler, see "Setting Up the C Compiler" on page 2-6.

Setting Up the C Compiler

Before using Embedded MATLAB MEX, you must set up your C compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference. You must run this command even if you use the default C compiler that comes with MATLAB for Windows platforms. You can also use `mex` to choose and configure a different C compiler, as described in "Compiler Requirements" in the MATLAB External Interfaces documentation.

Setting Up File Infrastructure and Paths

In this section...

Compile Path Search Order

Embedded MATLAB MEX resolves M-functions by searching first on the Embedded MATLAB path and then on the MATLAB path. See "How Embedded MATLAB Resolves Function Calls" on page 1-62 in the Embedded MATLAB documentation.

Can I Add Files to the Embedded MATLAB Path?

With Embedded MATLAB MEX, you can prepend directories and files to the Embedded MATLAB path, as described in "Adding Directories to Search Paths" on page 2-8. By default, the Embedded MATLAB path contains the current directory and the Embedded MATLAB libraries.

When to Use the Embedded MATLAB Path

Use the Embedded MATLAB path to override a MATLAB function with a customized version. Since Embedded MATLAB MEX searches the Embedded MATLAB path first, an M-file on the Embedded MATLAB path always shadows an M-file of the same name on the MATLAB path. To override a MATLAB function with a version implemented in Embedded MATLAB libraries, follow these steps:

1. Create each version of the M-function in identically-named M-files.

2. Add the MATLAB version to the MATLAB path.

3. Add the Embedded MATLAB version to the Embedded MATLAB path.

See “Adding Directories to Search Paths” on page 2-8.

Adding Directories to Search Paths

The following table explains how to add directories to search paths:

To add directories to:	Do this:
Embedded MATLAB path	Prepend directories to the Embedded MATLAB path using the compiler option `-I`. See “-I Add Directories to Embedded MATLAB Path” on page 4-4 using `emlmex`.
MATLAB path	Follow the instructions in “Adding a Directory to the Search Path” in the MATLAB Programming documentation.

Naming Conventions

Embedded MATLAB MEX enforces naming conventions for M-functions and generated files.

- “Reserved Prefixes” on page 2-8
- “Conventions for Naming Generated files” on page 2-8

Reserved Prefixes

Embedded MATLAB reserves the prefix `eml` for global C functions and variables in generated code. For example, Embedded MATLAB runtime library function names all begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C functions or primary M-functions with the prefix `eml`.

Conventions for Naming Generated files

Embedded MATLAB MEX follows MATLAB conventions by providing platform-specific extensions for C-MEX files.

Platform	MEX File Extension
`Linux (32-bit)`	`.mexglx`
`Linux x86-64`	`.mexa64`
`Windows (32-bit)`	`.mexw32`
`Window x64`	`.mexw64`

Making M-Code Compliant with Embedded MATLAB

In this section...
"Debugging Strategies" on page 2-10
"Detecting Embedded MATLAB Syntax Violations at Compile Time" on page 2-12

Debugging Strategies

Before performing code verification, The MathWorks recommends that you choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of M-files that call each other's functions. Here are two best practices:

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	1 Verify that your lowest-level (leaf) functions are compliant. 2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function.	• Efficient • Safe • Easy to isolate Embedded MATLAB syntax violations	Requires application tests that work from the bottom up
Top-down verification	1 Declare all functions called by the top-level function to be extrinsic so Embedded MATLAB MEX does not compile them (see "Declaring MATLAB Functions as Extrinsic Functions" on page 1-71). 2 Verify that your top-level function is compliant. 3 Work your way down the function hierarchy incrementally by removing extrinsic declarations one by one to compile and verify each function, ending with the leaf functions.	Lets you retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: • Extrinsic declarations • Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values (see "Working with mxArrays" on page 1-75)

Detecting Embedded MATLAB Syntax Violations at Compile Time

Before you can successfully convert an M-file to a C-MEX function, you must verify that your M-code complies with Embedded MATLAB syntax and semantics, as defined in Chapter 1, “Working with Embedded MATLAB”.

Embedded MATLAB MEX checks for all potential Embedded MATLAB syntax violations at compile time. When Embedded MATLAB MEX detects errors or warnings, it automatically generates a compilation report that describes the issues and provides links to the offending M-code. See “Working with Compilation Reports” on page 2-31.

If your M-code calls functions on the MATLAB path, Embedded MATLAB MEX attempts to compile these functions unless you declare them to be extrinsic (see “How Embedded MATLAB Resolves Function Calls” on page 1-62. To get detailed diagnostics, add the `%#eml` compiler directive to each external function that you want Embedded MATLAB MEX to compile, as described in “Adding the Compilation Directive %#eml” on page 1-67.

Specifying Properties of Primary Function Inputs

In this section...

Why You Must Specify Input Properties

Because C is a statically typed language, Embedded MATLAB MEX must determine the properties of all variables in the M-files at compile time. To infer variable properties in M-files, Embedded MATLAB MEX must be able to identify the properties of the inputs to the primary function, also known as the top-level or entry-point function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, also called *preconditions*, to Embedded MATLAB MEX. If your primary function has no input parameters, Embedded MATLAB MEX can compile your M-file without modification. You do not need to specify properties of inputs to subfunctions or external functions called by the primary function.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input:

For:	Specify Properties:				
	Class	**Size**	**Complexity**	**numerictype**	**fimath**
Fixed-point inputs	✓	✓	✓	✓	✓

<table>
<tr><th rowspan="2">For:</th><th colspan="5">Specify Properties:</th></tr>
<tr><th>Class</th><th>Size</th><th>Complexity</th><th>numerictype</th><th>fimath</th></tr>
<tr><td>Structure inputs</td><td colspan="5">Specify properties for each field according to its class
When a primary input is a structure, Embedded MATLAB MEX treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition, as follows:
• For each field of input structures, specify class, size, and complexity.
• For each field that is fixed-point class, also specify <code>numerictype</code>, and <code>fimath</code>.</td></tr>
<tr><td>All other inputs</td><td>✓</td><td>✓</td><td>✓</td><td></td><td></td></tr>
</table>

Default Property Values

Embedded MATLAB MEX assigns the following default values for properties of primary function inputs:

Property	Default
class	`double`
size	`scalar`
complexity	`real`
`numerictype`	No default
`fimath`	No default

Note In most cases, Embedded MATLAB MEX uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you may need to specify default values for properties of structure fields. For examples, see "Example: Specifying Class and Size of Scalar Structure" on page 2-26 and "Example: Specifying Class and Size of Structure Array" on page 2-27.

Supported Classes

The following table presents the class names supported by Embedded MATLAB MEX:

Class Name	Description
`logical`	Logical array of true and false values
`char`	Character array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`single`	Single-precision floating-point or fixed-point number array
`double`	Double-precision floating-point or fixed-point number array
`struct`	Structure array
`embedded.fi`	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

Follow these rules when specifying the properties of primary inputs:

- For each primary function input whose class is fixed point (`fi`), you must specify the input's `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

You can use any of the following methods to define the properties of primary function inputs:

Method	Pros	Cons
"Defining Input Properties by Example at the Command Line" on page 2-17	• Easy to use • Does not alter original M-code • Designed for prototyping a function that has a small number of primary inputs	• Must be specified at the command line every time you invoke Embedded MATLAB MEX (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Defining Input Properties Programmatically in the M-File" on page 2-19	• Integrated with M-code so you do not need to redefine properties each time you invoke Embedded MATLAB MEX • Provides documentation of property specifications in the M-code • Efficient for specifying memory-intensive inputs such as large structures	• Uses complex syntax

Note To specify the properties of inputs for any given primary function, use one of these methods or the other, but not both.

Defining Input Properties by Example at the Command Line

The command that invokes Embedded MATLAB MEX — `emlmex` — provides a command-line option `-eg` for specifying the properties of primary function inputs as a cell array of example values (see `emlmex`). The cell array can be a variable or literal array of constant values. Using this method, you specify the properties of inputs at the same time that you compile the M-file with Embedded MATLAB MEX.

- "Command Line Option -eg" on page 2-17
- "Rules for using the -eg option" on page 2-17
- "Examples: Specifying Properties of Primary Inputs by Example" on page 2-18
- "Examples: Specifying Properties of Primary Fixed-Point Inputs by Example" on page 2-18

Command Line Option -eg

The command that invokes Embedded MATLAB MEX — `emlmex` — provides a command-line option -eg for specifying the properties of primary function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you compile the M-file with Embedded MATLAB MEX. See "-eg Specify Input Properties by Example" on page 4-3 for `emlmex`.

Rules for using the -eg option

Follow these rules when using the `-eg` command-line option to define properties by example:

- The cell array of sample values must contain the same number of elements as primary function inputs.

- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Examples: Specifying Properties of Primary Inputs by Example

Consider an M-function that adds its two inputs:

```
function y = emcf(u,v)
y = u + v;
```

The following examples show how to specify different properties of the primary inputs u and v by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

  ```
  emlmex -o emcfx emcf -eg {0,0}
  ```

- Use a literal cell array of constants to specify that input u is an unsigned 16-bit, 1-by-4 vector and input v is a scalar double:

  ```
  emlc -o emcfx emcf -eg {zeros(1,4,'uint16'),0}
  ```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

  ```
  a = uint8([1;2;3;4])
  b = uint8([5;6;7;8])
  ex = {a,b}
  emlmex -o emcfx emcf -eg ex
  ```

Examples: Specifying Properties of Primary Fixed-Point Inputs by Example

Consider an M-function that calculates the square root of a fixed-point number:

```
function y = sqrtfi(x)
y = sqrt(x);
```

To specify the properties of the primary fixed-point input x by example on the MATLAB command line, follow these steps:

1 Define the numerictype properties for x, as in this example:

```
T = numerictype('WordLength',32,'FractionLength',23,'Signed',true);
```

2 Define the fimath properties for x, as in this example:

```
F = fimath('SumMode','SpecifyPrecision','SumWordLength',32,
           'SumFractionLength',23,'ProductMode','SpecifyPrecision',
           'ProductWordLength',32,'ProductFractionLength',23);
```

3 Create a fixed-point variable with the numerictype and fimath properties you just defined, as in this example:

```
myeg = { fi(4.0,T,F) };
```

4 Compile the function sqrtfi using the emlmex command, passing the variable myeg as the argument to the -eg option, as in this example:

```
emlmex sqrtfi -eg myeg;
```

Defining Input Properties Programmatically in the M-File

Embedded MATLAB MEX lets you use the MATLAB assert function to define properties of primary function inputs directly in your M-file.

- "How to Use assert with Embedded MATLAB MEX" on page 2-20
- "Rules for Using assert Function" on page 2-24
- "Example: Specifying General Properties of Primary Inputs" on page 2-25
- "Example: Specifying Properties of Primary Fixed-Point Inputs" on page 2-26
- "Example: Specifying Class and Size of Scalar Structure" on page 2-26
- "Example: Specifying Class and Size of Structure Array" on page 2-27

How to Use assert with Embedded MATLAB MEX

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

Specify Any Class (p. 2-20)	`assert (isa (param, 'class_name'))`
Specify fi Class (p. 2-21)	`assert (isfi (param))`
Specify Structure Class (p. 2-21)	`assert (isstruct (param))`
Specify Any Size (p. 2-22)	`assert (all (size(param == [dims]))`
Specify Scalar Size (p. 2-22)	`assert (isscalar((param))`
Specify Real Input (p. 2-23)	`assert (isreal((param))`
Specify Complex Input (p. 2-23)	`assert (~isreal((param))`
Specify numerictype of Fixed-Point Input (p. 2-23)	`assert (isequal (numerictype(fiparam), T))`
Specify fimath of Fixed-Point Input (p. 2-24)	`assert (isequal (fimath(fiparam),F))`
Specify Multiple Properties of Input (p. 2-24)	`assert (function1(params) && function2(params) && function3(params) && ...)`

Specify Any Class.

```
assert ( isa ( param, 'class_name') )
```

Sets the input parameter `param` to the MATLAB class *class_name*. For example, to set the class of input `U` to a 32-bit signed integer, call:

```
...
assert(isa(U,'int32'));
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype` and `fimath` properties (see "Specify numerictype of Fixed-Point Input" on page 2-23 and "Specify fimath of Fixed-Point Input" on page 2-24).

If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify fi Class.

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class `fi` (fixed-point numeric object). For example, to set the class of input `U` to `fi`, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U,'embedded.fi'));
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype` and `fimath` properties (see "Specify numerictype of Fixed-Point Input" on page 2-23 and "Specify fimath of Fixed-Point Input" on page 2-24).

Specify Structure Class.

```
assert ( isstruct ( param ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U,'struct'));
...
```

Note If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify Any Size.

```
assert ( all ( size (param == [dims ] ) )
```

Sets the input parameter *`param`* to the size specified by dimensions *`dims`*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size.

```
assert ( isscalar (param ) )
assert ( all ( size (param == [ 1 ] ) )
```

Sets the size of input parameter *`param`* to scalar. For example, to set the size of input `U` to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
```

```
assert(all(size(U)== [1]));
...
```

Specify Real Input.

```
assert ( isreal (param ) )
```

Specifies that the input parameter *param* is real. For example, to specify that input `U` is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input.

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *param* is complex. For example, to specify that input `U` is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input.

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the `numerictype` properties of `fi` input parameter *fiparam* to the `numerictype` object *T*. For example, to specify the `numerictype` property of fixed-point input `U` as a signed `numerictype` object `T` with 32-bit word length and 30-bit fraction length, use the following code:

```
...
% Define the numerictype object.
T = numerictype(1, 32, 30);

% Set the numerictype property of input U to T.
assert(isequal(numerictype(U),T));
...
```

Specify fimath of Fixed-Point Input.

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *`fiparam`* to the `fimath` object *`F`*. For example, to specify the `fimath` property of fixed-point input `U` so that it saturates on integer overflow, use the following code:

```
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

Specify Multiple Properties of Input.

```
assert ( function1 ( params ) && function2 ( params ) && function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
...
assert(isa(U,'double') && ~isreal(U) && all(size(U) == [3 3]) && isa(V,'uint16'));
...
```

Rules for Using assert Function

Follow these rules when using the `assert` function to specify the properties of primary function inputs:

- Call `assert` functions at the beginning of the primary function, before any flow-control operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- Use the `assert` function with Embedded MATLAB MEX only for specifying properties of primary function inputs before converting your M-code to C-MEX code.

- If you set the class of an input parameter to `fi`, you must also set its `numerictype` and `fimath` properties (see "Specify numerictype of Fixed-Point Input" on page 2-23 and "Specify fimath of Fixed-Point Input" on page 2-24).
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of each field in the structure in the order in which you define the fields in the structure definition.

Example: Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `emcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs:

Input	Property	Value
`pennywhistle`	class	`int16`
	size	220500-by-1 vector
	complexity	`real` (by default)
`win`	class	`double` (by default)
	size	1024-by-1 vector
	complexity	`real` (by default)

```
function y = emcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(all(size(win) == [nfft 1]));
...
```

Note If you do not specify the complexity of a primary function input, Embedded MATLAB MEX assumes it is `real` by default.

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands, as follows:

```
function y = emcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...
```

Example: Specifying Properties of Primary Fixed-Point Inputs

In the following example, the primary MATLAB function `emcsqrtfi` takes one fixed-point input: `x`. The code specifies the following properties for this input:

Property	Value
class	`fi`
`numerictype`	`numerictype` object `T`, as specified in the primary function
`fimath`	`fimath` object `F`, as specified in the primary function
size	`scalar` (by default)
complexity	`real` (by default)

```
function y = emcsqrtfi(x)
T = numerictype('WordLength',32,'FractionLength',23,
                'Signed',true);
F = fimath('SumMode','SpecifyPrecision',
           'SumWordLength',32,'SumFractionLength',23,
           'ProductMode','SpecifyPrecision',
           'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numerictype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Example: Specifying Class and Size of Scalar Structure

Assume you have defined `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

Here is code that specifies the class and size of S and its fields when passed as an input to your M-function:

```
function y = fcn(S)

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8'));
...
```

Note In most cases, Embedded MATLAB MEX uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S.r` is of type `double`, even though `double` is the default.

Example: Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined S as the following 2-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input S using the first element of the array:

```
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
```

```
% based on the first element of the array.
assert(all(size(S) == [2 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

Note In most cases, Embedded MATLAB MEX uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S(1).r` is of type `double`, even though `double` is the default.

Compiling Your M-File

In this section...

Running Embedded MATLAB MEX

You run Embedded MATLAB MEX from the MATLAB command prompt by using the `emlmex` function. The basic command is:

```
emlmex M_fcn
```

By default, `emlmex` performs the following actions:

- Searches for the function *M_fcn* stored in the file *M_fcn*.m as specified in "Compile Path Search Order" on page 2-7.
- Compiles *M_fcn*, checking for compliance with Embedded MATLAB.
- If there are no errors or warnings, generates a platform-specific C-MEX file in the current directory, using the naming conventions described in "Naming Conventions" on page 2-8.
- If there are errors, does not generate a C-MEX file, but produces an error report in a default output directory, as described in "Generated Files and Locations" on page 2-29. Error reports are described in "Working with Compilation Reports" on page 2-31.
- If there are warnings, but no errors, generates a platform-specific C-MEX file in the current directory, but does report the warnings.

You can modify this default behavior by specifying one or more compiler options with `emlmex`, separated by spaces on the command line. See "Options" on page 4-2 for `emlmex` in Chapter 4, "Functions — Alphabetical List".

Generated Files and Locations

By default, Embedded MATLAB MEX generates files in the following locations:

Generates:	In:
Platform-specific C-MEX files	Current directory
MAT-file (contains compilation information such as the name, size, and class of the M-code variables)	Default output directory: `emcprj/mexfcn/`*`M_fcn_name`*`/`*`M_fcn_name`*`_report.mat`
HTML reports (if errors or warnings occur during compilation)	Default output directory: `emcprj/mexfcn/`*`M_fcn_name`*`/html`

You can change the name and location of generated files by using the options `-o` and `-d` when you run Embedded MATLAB MEX (see `emlmex`).

Working with Compilation Reports

In this section...

About Compilation Reports

Embedded MATLAB MEX automatically generates reports in HTML format when errors or warnings occur at compile time. You can use these reports for debugging your M-code and verifying compliance with Embedded MATLAB.

Location of Compilation Reports

Embedded MATLAB MEX describes errors and warnings in HTML reports at the following location:

```
output_directory/mexfcn/
primary_function_name/html/
primary_function_name_report.html
```

Note The default output directory is `emcprj`, but you can specify a different directory with the `-d` option (see `emlmex`).

Description of Compilation Reports

Embedded MATLAB MEX automatically reports errors and warnings. If errors occur during compilation, Embedded MATLAB MEX does not generate C-MEX code. If compilation produces warnings, but no errors, Embedded MATLAB MEX does generate C-MEX code, but displays the warning messages.

Reports present error and warning messages in two views:

- List view (default)

For example:

Type	Location	Message
Warning	emcsincos.m:2:1	Unsupported: GLOBAL variables.
Error	emcsincos.m:6:1	Unsupported: GLOBAL variables.
Error	emcsincos.m:7:1	Function 'plot' resolved in the MATLAB workspace. Implicit evaluation in MATLAB is not supported. Please declare this function extrinsic using eml.extrinsic('plot'), or call it using feval.

- Tree view

  For example:

In either view, the **Location** link brings you to the offending code in the source file, while the **Message** link highlights the location of the offending code in the source listing, as in this example:


Type
Location
Message
Warning
emcsincos.m:2:1
Unsupported: GLOBAL variables.
Error
emcsincos.m:6:1
Unsupported: GLOBAL variables.
Error
emcsincos.m:7:1
Function 'plot' resolved in the MATLAB workspace.
not supported.
Highlight the error line in this report
sic using
eml.extrinsic('plot'), or call it using feval.
Function emcsincos
function r = emcsincos(num)
global x;
assert(isa(num,'double'));
assert(all(size(num) == [1 10]));
r = sin(num);
x = cos(num);
plot(num,r);

Calling C Functions from Embedded MATLAB

The eml C Interface

In this section...

What Is the eml C Interface?

The `eml` C interface provides a method for calling external C functions from Embedded MATLAB functions using `eml` functions. In addition to using the `eml` functions, you need to include C libraries, object files, and header files to configure your environment.

The `eml` C interface lets you:

- Use legacy C code in Embedded MATLAB.
- Use your own optimized C functions instead of generated code.
- Interface your libraries and hardware with Embedded MATLAB.

The `eml` interface provides unrestricted access to arbitrary C code. Misuse of the interface or errors in your C code can crash or destabilize MATLAB.

Calling External C Functions

Use the `eml.ceval` function to call external C functions from an Embedded MATLAB function. `eml.ceval` passes function input and output arguments to C functions by value or reference. You must define C functions called by Embedded MATLAB functions in external C source files or in C libraries, not in the model that contains the Embedded MATLAB function.

Passing Arguments by Reference to External C Functions

The `eml` interface provides the following constructs that allow you to pass Embedded MATLAB variables as arguments by reference to external C functions:

- `eml.ref` — pass value by reference
- `eml.rref` — pass read-only value by reference
- `eml.wref` — pass write-only value by reference

These constructs offer the following benefits:

- Passing values by reference optimizes memory use.

  When you pass arguments by value, Embedded MATLAB passes a copy of the value of each argument to the C function to preserve the original values. When you pass arguments by reference, Embedded MATLAB does not copy values. The memory savings can be significant if you need to pass large matrices to the C function.

- Passing write-only values by reference returns multiple outputs.

  By using `eml.wref`, you can achieve the effect of returning multiple outputs from your C function, including arrays and matrices (see “Returning Multiple Values from C Functions” on page 3-11). Otherwise, the C function can return only a single scalar value through its `return` statement.

When you pass arguments by reference using `eml.rref`, `eml.wref`, and `eml.ref`, the corresponding C function signature must declare these variables as pointers of the same data type. Otherwise, the C compiler generates a type mismatch error.

Do not store pointers that you pass to C functions because the results can be unpredictable. For example, if Embedded MATLAB passes a pointer to an array using `eml.ref`, `eml.rref`, or `eml.wref`, then the C function can modify the data in the array—but you should not store the pointer for future use.

For example, suppose your model contains an Embedded MATLAB function that calls an external C function `ctest`:

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

y = 0;
y = eml.ceval('ctest',u);
```

Now suppose the C function signature is:

```
real32_T ctest(real_T *a)
```

When you compile the model, you get a type mismatch error because `eml.ceval` calls `ctest` with an argument of type `double` when `ctest` expects a pointer to a double-precision, floating-point value.

Match the types of arguments in `eml.ceval` with their counterparts in the C function. For instance, you can fix the error in the previous example by passing the argument by reference:

```
y = eml.ceval('ctest', eml.rref(u));
```

You can pass a reference to an element of a matrix. For example, to pass the second element of the matrix `v`, you can use the following code:

```
y = eml.ceval('ctest', eml.ref(v(1,2));
```

Declaring Data

The `eml` interface provides the construct `eml.opaque` that allows you to manipulate C data that Embedded MATLAB does not understand. You can store the opaque data in a variable or structure field and pass it to, or return it from, a C function using `eml.ceval`.

Calling External C Functions

In this section...

Workflow for Calling External C Functions

To call external C functions from Embedded MATLAB using the `eml` interface:

1. Write your C functions in external source files or libraries.

2. Create header files, if necessary.

   The header file defines the data types used by the C functions that Embedded MATLAB generates in code, as described in "Mapping MATLAB Types to C" on page 3-15.

   Tip One way to add these type definitions is to include the header file `tmwtypes.h`, which defines all general data types supported by MATLAB. This header file is in *`matlabroot`*`/extern/include`. If you plan to generate Real-Time Workshop code, check the definitions in `tmwtypes.h` to determine if they are compatible with your Real-Time Workshop target. If not, define these types in your own header files.

   ---

3. In your Embedded MATLAB script, add calls to `eml.ceval` to invoke your external C functions.

   You must add one `eml.ceval` statement for each C function that you wish to make. In your `eml.ceval` statements, use `eml.ref`, `eml.rref`, and `eml.wref` constructs as needed (see "Passing Arguments by Reference to External C Functions" on page 3-3).

4. Include the custom C functions in generated code (see "Including Custom C Functions in Generated Code" on page 3-7).

5. Configure your C compiler to treat warnings as errors (optional).

This step ensures that you catch type mismatches between C and Embedded MATLAB (see “How Embedded MATLAB Infers C Data Types” on page 3-15).

6 Build your model and fix errors.

7 Run your model.

eml Custom C Interface Best Practices

The following are recommended practices when using the `eml` interface.

- **Start small.** — Create a test function and learn how the `eml` interface works.
- **Use separate files.** — Create an M-file for each C function that you call. Make sure that each call to the C function has the correct type.

Including Custom C Functions in Generated Code

In this section...

Including C Functions in Simulation Targets

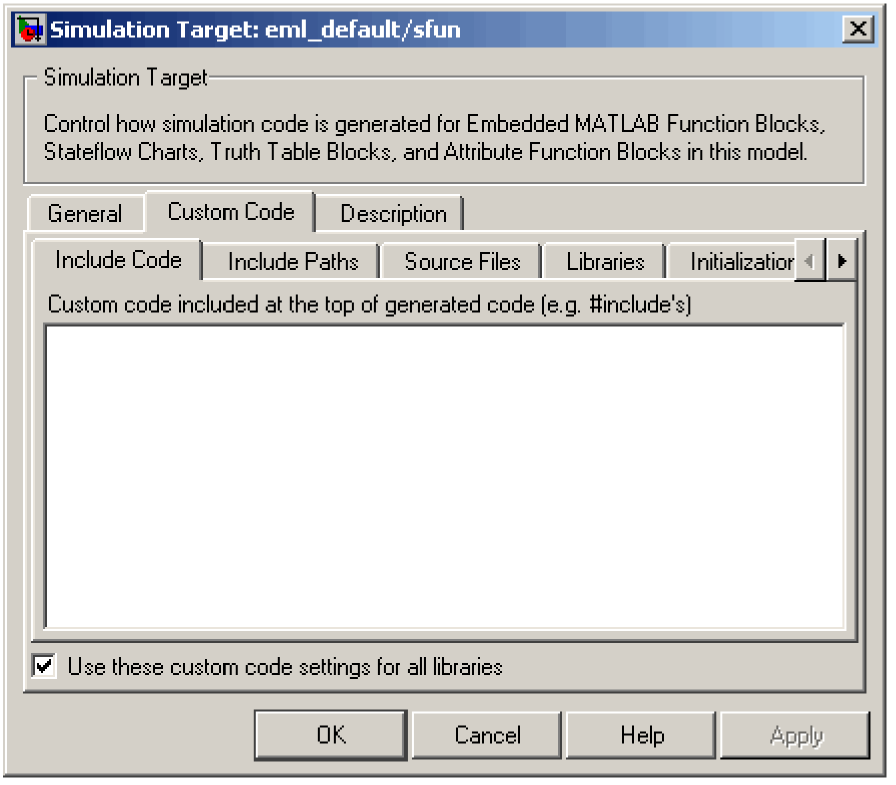
To include the C functions that you call from Embedded MATLAB functions in code generated for simulation, follow these steps:

1. Open the Embedded MATLAB editor.

2. Click the Simulation Target icon.

   The Simulation Target dialog opens.

3 In the Simulation Target dialog, click the **Custom Code** tab.

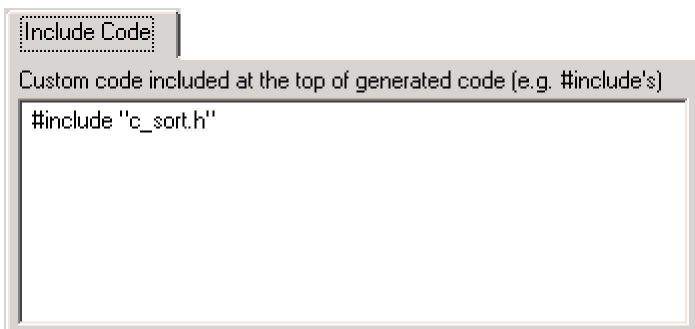


4 In the Simulation Target dialog, enter the following information:

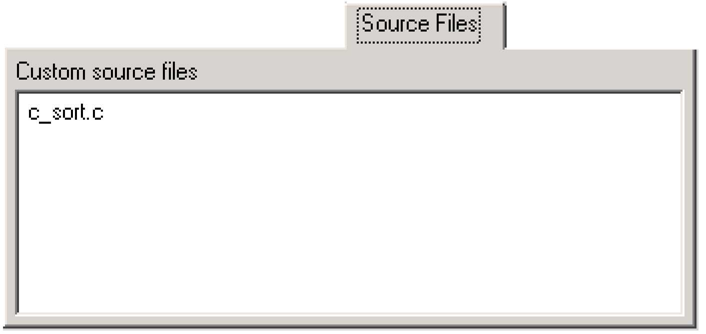
To include:	Enter:	Where?
C header files associated with your custom C functions	Names of one or more header files, one on each line, in this format: `#include "file_name.h"`	**Include Code** tab

To include:	Enter:	Where?
Paths to the external C header files and source code	One or more relative paths, separated by commas, spaces, or new lines **Note** Paths should be relative to the directory where your model resides.	**Include Paths** tab
Source files for your custom C functions	Names of one or more C source files, separated by commas, spaces, or new lines	**Source Files** tab
Libraries	Names of one or more static libraries containing your custom object code, separated by commas, spaces, or new lines	**Libraries** tab

For example, suppose your Embedded MATLAB function calls a C function with the prototype declared in `c_sort.h` and defined in `c_sort.c`. Both files reside in the same directory as the parent model. To include the C header file in generated code, enter this information:



To include the source file in generated code, enter this information:



5 Click **OK**.

Including C Functions in Real-Time Workshop Targets

To generate code for real-time applications, you must have licenses for Real-Time Workshop and, optionally, for Real-Time Workshop Embedded Coder for production applications. For information about licensing, contact MathWorks Technical Support: `www.mathworks.com/support`.

To include your custom C functions in code generated by Real-Time Workshop:

- For Simulink models, see "Configuring Custom Code" in the Real-Time Workshop User's Guide.
- For MATLAB code, see "Working with Embedded MATLAB Coder" in the Real-Time Workshop User's Guide.

Code Examples

In this section...

Returning Multiple Values from C Functions

The C language restricts functions from returning multiple outputs; instead, they return only single, scalar values. The constructs `eml.ref` and `eml.wref` allow Embedded MATLAB functions to exchange multiple outputs with the external C functions that they call.

For example, suppose you write an Embedded MATLAB function `foo` that takes two inputs `x` and `y`, and returns three outputs `a`, `b`, and `c`. In Embedded MATLAB, you call this function as follows:

```
[a, b, c] = foo (x, y)
```

If you rewrite `foo` as a C function, you cannot return `a`, `b`, and `c` through the `return` statement. You can create C functions with multiple pointer type input arguments, and pass the output parameters by reference. Then you can call the C functions with multiple outputs from Embedded MATLAB functions using `eml.wref` constructs:

```
eml.ceval ('foo', eml.rref(x), eml.rref(y), ...
    eml.wref(a), eml.wref(b), eml.wref(c));
```

Similarly, suppose that one of the outputs `a` is also an input argument. In this case, you call the Embedded MATLAB function `foo` as in this example:

```
[a, b, c] = foo (a)
```

To call a C version of `foo` with the same arguments from Embedded MATLAB, use `eml.wref` and `eml.rref` constructs:

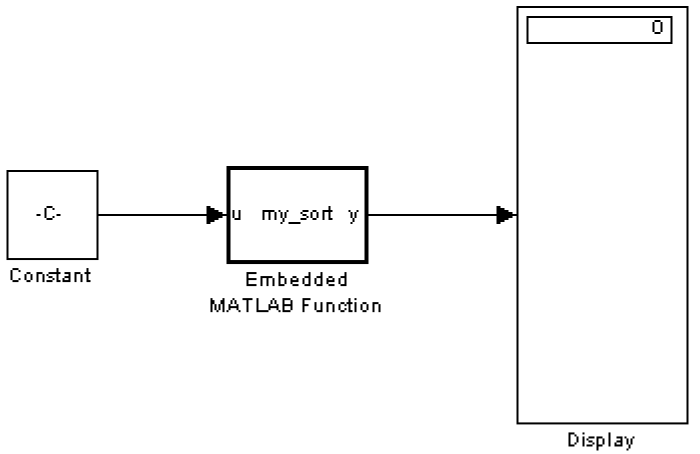
```
eml.ceval ('foo', eml.rref(a), eml.wref(a),
            eml.wref(b), eml.wref(c));
```

Calling an External C Sort Function

This example shows how to call a custom C sort function from an Embedded MATLAB Function block in a Simulink model.

C Sort Example: The Simulink Model

The Simulink model that calls the C sort function looks like this:



This model contains an Embedded MATLAB Function block with one input u and one output y. Input u is an unsorted vector constant; output y is the sorted vector. The Embedded MATLAB function calls a custom C function to sort the vector:

```
function y = my_sort(u)

% Constrain y to the same size and class as u
y = u;

% Identify the MATLAB function 'disp' as an extrinsic function
eml.extrinsic('disp');

% Constrain the return type of C function c_sort
%  to a logical value
s = false;
```

```
% Call 'c_sort' with arguments:
%     numel(y)    - The number of elements of y
%     eml.rref(y) - A read only reference to input y
%     eml.wref(y) - A write only reference to output y
s = eml.ceval('c_sort', eml.rref(y),
              eml.wref(y), int32(numel(y)));

if s % Already sorted
    disp('Already sorted.');
end
```

Note In this Embedded MATLAB code:

- To optimize memory allocation, Embedded MATLAB uses the same memory location to store the unsorted input vector `u` and the sorted output vector `y`. Passing `eml.rref(y)` and `eml.wref(y)` requires the C function to read from and write to the memory location referenced by `y`. The memory can be shared because `u` is the same type and size as `y`, and because the C function uses an algorithm that sorts in place.
- Embedded MATLAB cannot infer the types of the output `y` or the C function return value `s`, so you must set them explicitly before calling `eml.ceval`.

C Sort Example: The Custom C Code

The Embedded MATLAB function calls a C function that uses a custom C header file. The header file `c_sort.h` contains the following code:

```
#include <tmwtypes.h>

boolean_T c_sort(real_T *U, real_T *Y, int32_T n);
```

Note In this header file, `tmwtypes.h` defines all general data types supported by Embedded MATLAB (see "Calling External C Functions" on page 3-5).

The function `c_sort.c` contains the following code:

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

static int my_cmp(void *A, void *B)
{
    real_T A1 = *((real_T *)A);
    real_T B1 = *((real_T *)B);

    if (A1 < B1) return -1;
    if (A1 > B1) return 1;
    return 0;
}

boolean_T c_sort(real_T *U, real_T *Y, int32_T n)
{
    boolean_T sorted = 1;
    int i;
    for (i = 0; i < n-1; i++) {
        if (U[i] > U[i+1]) sorted = 0;
    }

    for (i = 0; i < n; i++) {
        Y[i] = U[i];
    }

    if (~sorted) {
        qsort(Y, n, sizeof(real_T), my_cmp);
    }

    return sorted;
}
```

Note In this C function, the statement `Y[i] = U[i]` ensures that the C function writes to all elements of Y, a requirement when you pass the variable through `eml.wref` to the C function (see `eml.wref`).

How Embedded MATLAB Infers C Data Types

In this section...

Mapping MATLAB Types to C

The C type associated with an Embedded MATLAB variable or expression is based on the following properties:

- Class
- Size
- Complexity

The following translation table shows the MATLAB types supported by Embedded MATLAB, and how Embedded MATLAB infers the generated code types.

MATLAB Type	C Type	C Reference Type
int8	int8_T	int8_T *
int16	int16_T	int16_T *
int32	int32_T	int32_T *
uint8	uint8_T	uint8_T *
uint16	uint16_T	uint16_T *
uint32	uint32_T	uint32_T *
double	real_T	real_T *
single	real32_T	real32_T *

MATLAB Type	C Type	C Reference Type
char	char	char *
logical	boolean_T	boolean_T *
fi	numericaltype also influences the C type. Integer type varies according to the Embedded MATLAB fixed-point type, as described in"Mapping embedded.numerictypes to C" on page 3-16.	
struct	Fields also affect the C type. See "Mapping Structures to C Structures" on page 3-18.	
complex	See "Mapping embedded.numerictypes to C" on page 3-16.	
function handles	Not supported.	

Mapping embedded.numerictypes to C

The following translation table shows how Embedded MATLAB infers integer types from fixed-point objects. In the first column, the fixed-point types are specified by the Fixed-Point Toolbox numerictype function:

```
numerictype(signedness, word length, fraction length)
```

The Embedded MATLAB integer type is the next larger target word size that can store the fixed-point value, based on its word length. The sign of the integer type matches the sign of the fixed-point type.

embedded.numerictype	C Type	C Reference Type
numerictype(1, 16, 15)	int16_T	int16_T *
numerictype(1, 13, 10)	int16_T	int16_T *
numerictype(0, 19, 15)	uint32_T	uint32_T *
numerictype(1, 8, 7)	int8_T	int8_T *

Mapping Arrays to C

The following translation table shows how Embedded MATLAB determines array types and sizes in generated code. In the first column, the arrays are specified by the MATLAB `zeros` function:

```
zeros(number of rows, number of columns, data type)
```

Embedded MATLAB array data is laid out in column major order.

Array	C Type	C Reference Type
`zeros(10, 5, 'int8')`	`int8_T *`	`int8_T *`
`zeros(5, 10, 'int8')`	`int8_T *`	`int8_T *`
`zeros(3, 7)`	`real_T *`	`real_T *`
`zeros(10, 1, 'single')`	`real32_T *`	`real32_T *`

Mapping Complex Values to C

The following translation table shows how Embedded MATLAB infers complex values in generated code.

Complex	C Type	C Reference Type
`complex int8`	`cint8_T`	`cint8_T *`
`complex int16`	`cint16_T`	`cint16_T *`
`complex int32`	`cint32_T`	`cint32_T *`
`complex uint8`	`cuint8_T`	`cuint8_T *`
`complex uint16`	`cuint16_T`	`cuint16_T *`
`complex uint32`	`cuint32_T`	`cuint32_T *`
`complex double`	`creal_T`	`creal_T *`
`complex single`	`creal32_T`	`creal32_T *`

Embedded MATLAB defines each complex value as a structure with a real component `re` and an imaginary component `im`, as in this example from `tmwtypes.h`:

```
typedef struct {
  real32_T re;/* Real component*/
  real32_T im;/* Imaginary component*/
} creal32_T;
```

Embedded MATLAB uses the names `re` and `im` in generated code to represent the components of complex numbers. For example, suppose you define a variable `x` of type `creal32_T`. The generated code references the real component as `x.re` and the imaginary component as `x.im`.

If your C library requires a different representation, you can define your own versions of Embedded MATLAB complex types, but you *must* use the names `re` for the real components and `im` for the imaginary components in your definitions.

Embedded MATLAB represents a matrix of complex numbers as an array of structures.

Mapping Structures to C Structures

Embedded MATLAB translates structures to C types field-by-field. The order of the field items is preserved as given in MATLAB. To control the name of the generated C structure type, or provide a definition, use the `eml.cstructname` function.

Note Arrays in structures translate into single-dimension arrays, not pointers.

Mapping Strings to C

Embedded MATLAB translates MATLAB strings to C character matrices. Character matrices cannot be used as substitutes for C strings because they are not null terminated. You can terminate a MATLAB string with a null character by appending a zero to the end of the string: `['sample string' 0]`. A single character translates to a C `char` type, not a C string.

Caution Failing to null-terminate your MATLAB strings causes C code to crash without compiler errors or warnings.

4

Functions — Alphabetical List

emlmex

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 4-2.

Options

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

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

- "-? Display Help" on page 4-5

-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, Embedded MATLAB MEX creates them for you. *`out_directory`* can be an absolute path or relative path. If you do not specify an output directory, Embedded MATLAB MEX 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" on page 2-17.

-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 `fimath` function, as in this example:

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

Embedded MATLAB MEX 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" on page 2-17) or programmatically (see "Defining Input Properties Programmatically in the M-File" on page 2-19). If you do not define a default value, then you must use one of the other methods to specify the `fimath` property of your primary, fixed-point inputs.

-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 .

-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 `numerictype` function, 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" on page 2-17) or programmatically (see "Defining Input Properties Programmatically in the M-File" on page 2-19). 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).

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
`disable:inline`	Disable function inlining.
`enable:inline`	Enable function inlining (default).

-? 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 4-6.

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 .

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) 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" on page 2-19.

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

Evaluate external C function

Syntax

`[y =] eml.ceval('function_name', u1 ..., un);`

Arguments

function_name
External C function to execute.

u_1 ..., u_n
Expressions or function calls using `eml.rref`, `eml.wref`, and `eml.ref`.

Description

`[y =] eml.ceval('function_name', u1 ..., un);` executes the external C function specified by the quoted string *function_name* with optional arguments u_1 ..., u_n. You must define the C function *function_name* in an external C source file or library. The *function_name* parameter cannot contain path information.

Optionally, `eml.ceval` can return a single scalar value, corresponding to the value returned by the C function in the `return` statement. To be consistent with C, `eml.ceval` cannot return a vector or matrix.

Note To allow Embedded MATLAB to infer the data type of return values and output arguments, you must constrain their values before calling `eml.ceval` (see "Example" on page 4-8).

Caution

`eml.ceval` is an Embedded MATLAB only function. Using it in MATLAB generates an error.

Example

Assume that you want to call the C function `foo(u)`, which takes an input of type `real_T` and returns a value of type `int32`. To call this C function from an Embedded MATLAB function, use the following code:

```
y = int32(0); %Constrain the return type to int32_T
y = eml.ceval('foo', 10);
```

For this function call, Real-Time Workshop generates the following code:

```
int32_T eml_y;
eml_y = foo(10.0);
```

In this example, Embedded MATLAB cannot infer the type of the return value because it is defined in the C function. Therefore, you must constrain the return type to match the C function. Mismatched types are detected by the C compiler and generate compiler errors.

The type of the argument—in this case, the constant `10`—is not specified. Therefore, the argument implicitly defaults to double-precision, floating-point type, because the default data type in MATLAB is `double`. To specify a type explicitly, cast the argument, as in this example:

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

In this case, the generated code for the function call looks like this:

```
int32_T eml_y;
eml_y = foo(10);
```

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.rref`, `eml.wref`, and `eml.ref` (see "Passing Arguments by Reference to External C Functions" on page 3-3). If C does not support passing arguments by value, `eml.ceval` passes arguments by reference. In this case, if the `eml.ref`, `eml.rref`, and `eml.wref` constructs are not used, a copy of the argument is introduced.

See Also

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

eml.cstructname

Purpose Specify structure name in generated code

Syntax `eml.cstructname(var, 'name', 'extern')`

Arguments

var
: Structure variable.

name
: Specifies name to use for the structure.

`'extern'` (optional)
: Declares an externally defined structure. Embedded MATLAB does not generate the definition of the structure type; it must be provided in a custom include file.

Description `eml.cstructname(var, 'name', 'extern')` allows you to specify the name of a structure in generated code.

Note `eml.cstructname` has no effect in MATLAB; it applies to Embedded MATLAB only.

Example The following code is used by Embedded MATLAB to assign the name `MyPointType` to the structure `var`.

```
% Declare a MATLAB structure.
var.x = 1;
var.y = 2;

% Instruct Embedded MATLAB to assign the name MyPointType
% to the type of var.
eml.cstructname(var, 'MyPointType', 'extern');

% The type of var matches foo's signature.
eml.ceval('foo', eml.rref(var));
```

The corresponding C code looks like this:

```
typedef struct { x, y: double; } MyPointType;
void foo(const MyPointType *ptr);
```

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 a function executed by MATLAB during simulation. Embedded MATLAB 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.

Caution

`eml.extrinsic` is an Embedded MATLAB only function. Using it in MATLAB generates an error.

Example

The following code declares the MATLAB `find` function extrinsic in the Embedded MATLAB `foo` function:

```
function y = foo

eml.extrinsic('find');

x = ones(4);
y = x;
y = find(x);
```

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 Embedded MATLAB does not understand. *type* and *value* are treated as strings by Embedded MATLAB. 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 in32 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`

eml.ref

Purpose Pass argument by reference as read input or write output

Syntax `[y =] eml.ceval('function_name', eml.ref(arg), ... u_n)`

Arguments

arg
: Variable passed by reference as an input or an output to the external C function called in `eml.ceval`.

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

Caution

`eml.ref` is an Embedded MATLAB only function. Using it in MATLAB generates an error.

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, 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" on page 3-15.

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`*`)`**`, ... `u_n`)`

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`*`)`**`, ... `u_n`)` 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.
- `eml.ref` is an Embedded MATLAB only function. Using it in MATLAB generates an error.

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" on page 3-15.

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

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

- Embedded MATLAB 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.
- `eml.wref` is an Embedded MATLAB only function. Using it in MATLAB generates an error.

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.
- Embedded MATLAB 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" on page 3-15.
- Embedded MATLAB collapses matrices to a single dimension in the generated C code.

See Also

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

Index