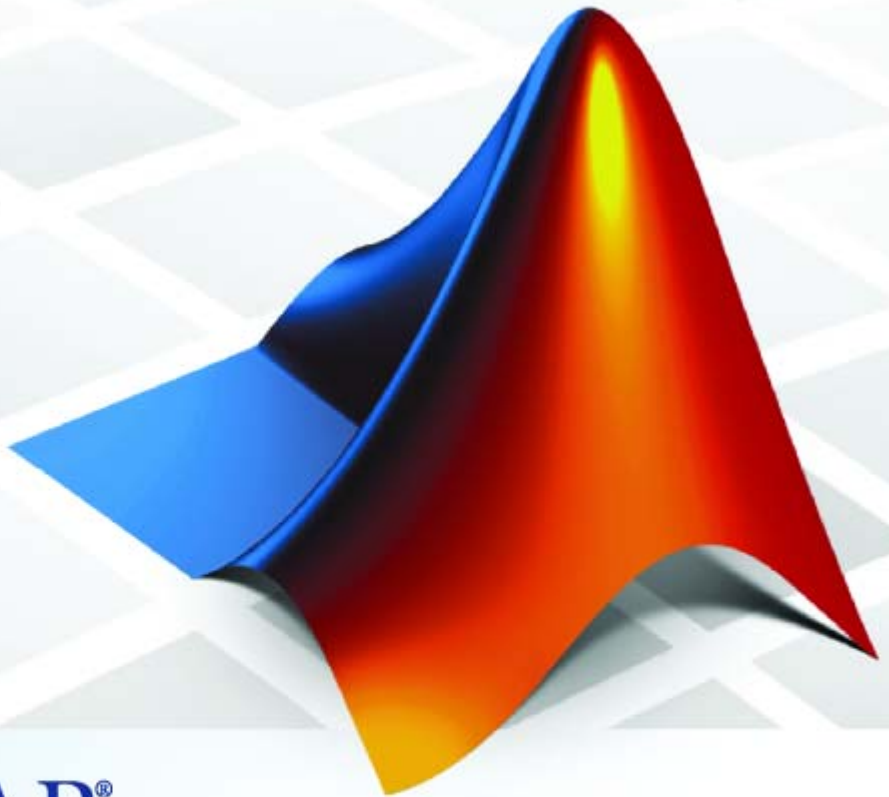


Embedded MATLAB 1

User's Guide



MATLAB®

How to Contact The MathWorks



www.mathworks.com
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab@mathworks.com)
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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

© COPYRIGHT 2007 by The MathWorks, Inc.

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

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

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

March 2007 Online only New for Version 1.0 (Release 2007a)

1

Working with Embedded MATLAB

What Is Embedded MATLAB?	1-2
Embedded MATLAB Inference Engine	1-2
Supported MATLAB Features	1-2
Unsupported MATLAB Features	1-3
Feature Limitations	1-4
 Using Embedded MATLAB with MathWorks	
Products	1-7
 Working with Variables	1-9
Supported Variable Types	1-9
Local Variables	1-9
 Operators	1-14
Control Flow Statements	1-14
Arithmetic Operators	1-15
Relational Operators	1-16
Logical Operators	1-16
 Embedded MATLAB Run-Time Function Library	1-18
Embedded MATLAB Run-Time Function Library —	
Alphabetical List	1-18
Embedded MATLAB Run-Time Library — Categorical	
List	1-58
 Calling Functions in Embedded MATLAB	1-78
How Embedded MATLAB Resolves Function Calls	1-78
Calling Subfunctions	1-80
Calling Embedded MATLAB Run-Time Library	
Functions	1-80
Calling MATLAB Functions	1-81
 Using Structures	1-90

About Embedded MATLAB Structures	1-90
Elements of Embedded MATLAB Structures	1-90
Types of Embedded MATLAB Structures	1-91
Defining Local Structure Variables	1-92
Defining Outputs as Structures	1-95
Making Structures Persistent	1-96
Indexing SubStructures and Fields	1-96
Assigning Values to Structures and Fields	1-97
Limitations with Structures	1-97
Using Function Handles	1-102
Example: Defining and Passing Function Handles in an Embedded MATLAB Function	1-102
Limitations with Function Handles	1-104
Using M-Lint with Embedded MATLAB	1-106

Working with Embedded MATLAB MEX

2

About Embedded MATLAB MEX	2-2
Running a Demo for Embedded MATLAB MEX	2-2
Workflow for Converting M-Code to a C-MEX Function	2-4
Installing Embedded MATLAB MEX	2-5
Setting Up the C Compiler	2-6
Setting Up File Infrastructure and Paths	2-7
Embedded MATLAB Path	2-7
Maintaining Two Versions of M-Code	2-9
Maintaining One Version of M-Code	2-9
Verifying M-Code Compliance with Embedded MATLAB 	2-10

Code Verification Methods	2-10
Debugging Strategies	2-12
Reserved Prefixes	2-13
Specifying Properties of Primary Function Inputs	2-15
Properties to Specify	2-15
Rules for Specifying Properties of Primary Inputs	2-17
Methods for Defining Properties of Primary Inputs	2-17
Defining Input Properties by Example at the Command Line	2-18
Defining Input Properties Programmatically in the M-File	2-21
Compiling Your M-File	2-30
Running Embedded MATLAB MEX	2-30
Generated Files and Locations	2-31
File Naming Conventions	2-31
Working with Compilation Reports	2-32
Location of Compilation Reports	2-32
Description of Compilation Reports	2-32

Functions — Alphabetical List

3

Index

Working with Embedded MATLAB

What Is Embedded MATLAB? (p. 1-2)	Describes the Embedded MATLAB subset of the MATLAB® language
Using Embedded MATLAB with MathWorks Products (p. 1-7)	Describes how to use Embedded MATLAB with Simulink®, Stateflow®, and Fixed-Point Toolbox
Working with Variables (p. 1-9)	Explains how to use variables in Embedded MATLAB and lists the supported types
Operators (p. 1-14)	Lists operators supported by Embedded MATLAB functions
Embedded MATLAB Run-Time Function Library (p. 1-18)	Lists run-time library functions that you can call in an Embedded MATLAB function
Calling Functions in Embedded MATLAB (p. 1-78)	Presents rules for calling functions in Embedded MATLAB and using their return values
Using Structures (p. 1-90)	Explains how to define and use structures in Embedded MATLAB
Using Function Handles (p. 1-102)	Describes how to use function handles in Embedded MATLAB
Using M-Lint with Embedded MATLAB (p. 1-106)	Explains how Embedded MATLAB automatically checks code with M-Lint

What Is Embedded MATLAB?

Embedded MATLAB is a restricted subset of the MATLAB language that provides optimizations for generating efficient, production-quality C code for embedded applications and accelerating fixed-point algorithms.

This topic describes the features of Embedded MATLAB.

- “Embedded MATLAB Inference Engine” on page 1-2
- “Supported MATLAB Features” on page 1-2
- “Unsupported MATLAB Features” on page 1-3
- “Feature Limitations” on page 1-4

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 “Limitations on Indexing Operations” on page 1-4)
- Complex numbers (see “Limitations with Complex Numbers” on page 1-5)
- Numeric classes (see “Supported Variable Types” on page 1-9)
- 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 1-78)
- Persistent variables (see “Declaring Persistent Variables” on page 1-13)
- Structures (see “Using Structures” on page 1-90)
- Characters (see “Limitations with Characters” on page 1-6)
- Function handles (see “Using Function Handles” on page 1-102)
- Frames (see “Working with Frame-Based Signals” in the Simulink documentation.
- Subset of MATLAB functions (see “Embedded MATLAB Run-Time Function Library” on page 1-18)

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 of dynamic size, or variables of different sizes.

- Global variables
- Java
- MATLAB functions

Note Embedded MATLAB does not generate code for functions on the MATLAB path, but allows you to call MATLAB functions during simulation (see “Calling MATLAB Functions” on page 1-81).

- Matrix deletion
- Nested functions
- Objects
- Sparse matrices
- Try/catch statements

Feature Limitations

This topic describes limitations on MATLAB features supported by Embedded MATLAB .

- “Limitations on Indexing Operations” on page 1-4
- “Limitations with Complex Numbers” on page 1-5
- “Limitations with Characters” on page 1-6

Limitations on 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:

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

- $M(i:i+k)$ 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 $0 \dots k$. 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.

Limitations with Complex Numbers

Embedded MATLAB supports complex numbers and operations 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)));
```

Limitations 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.

Using Embedded MATLAB with MathWorks Products

Embedded MATLAB works with the following products:

Product	Interface with Embedded MATLAB	Details
Simulink	<p>Provides</p> <ul style="list-style-type: none">• The front-end for writing and simulating Embedded MATLAB functions in Simulink models• Embedded MATLAB MEX as the back end for generating C-MEX functions from M-code that complies with Embedded MATLAB	<p>See:</p> <ul style="list-style-type: none">• “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	<p>Provides the front-end for writing and simulating Embedded MATLAB functions in Stateflow charts.</p>	<p>See “Using Embedded MATLAB Functions” and “Truth Table Functions” in the Stateflow documentation.</p>

Product	Interface with Embedded MATLAB	Details
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.
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.

Working with Variables

This topic describes how to create local and persistent variables in Embedded MATLAB.

- “Supported Variable Types” on page 1-9
- “Local Variables” on page 1-9

Supported Variable Types

Embedded MATLAB functions support a subset of MATLAB data types represented by the following cast 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 1-90)
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 Implicitly” on page 1-10
- “Creating Local Complex Variables Implicitly” on page 1-11
- “Declaring Persistent Variables” on page 1-13

Creating Local Variables Implicitly

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 1-13.
- 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
```

Creating Local Complex Variables Implicitly

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 the variable to a complex constant 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 a variable is typed and sized, it cannot be cast 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.
```

Declaring Persistent Variables

Persistent variables are local to the function in which they are declared, but their values are retained 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;
```

Operators

Embedded MATLAB functions support a subset of MATLAB operators.

- “Control Flow Statements” on page 1-14
- “Arithmetic Operators” on page 1-15
- “Relational Operators” on page 1-16
- “Logical Operators” on page 1-16

Each listing includes a link to the MATLAB Function Reference documentation (help) for the equivalent MATLAB function along with a one-line description and any limitations that apply.

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

Statement	Description
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:

Operation	Description
&	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.
	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.

Operation	Description
-	Element complement
xor	Logical exclusive-OR
&&	Logical AND (short-circuiting)
	Logical OR (short-circuiting)

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

Embedded MATLAB Run-Time Function Library

This section lists the MATLAB functions supported by Embedded MATLAB in its library of run-time functions. Each Embedded MATLAB library function has the same name, arguments, and functionality as its MATLAB, Fixed-Point Toolbox, or Signal Processing Toolbox counterpart, but they 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 use the generated code to build a portable standalone executable target.

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.

The following topics list and describe the functions supported by the Embedded MATLAB run-time library:

- “Embedded MATLAB Run-Time Function Library — Alphabetical List” on page 1-18
- “Embedded MATLAB Run-Time Library — Categorical List” on page 1-58

Embedded MATLAB Run-Time Function Library — Alphabetical List

This topic lists the MATLAB functions supported by Embedded MATLAB in alphabetical order. See also “Embedded MATLAB Run-Time Library — Categorical List” on page 1-58.

Function	Product	Remarks/Limitations
abs	MATLAB	—
abs	Fixed-Point Toolbox	—
acos	MATLAB	<ul style="list-style-type: none">• Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.

Function	Product	Remarks/Limitations
acosd	MATLAB	—
acosh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
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	—
asin	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
asind	MATLAB	—

Function	Product	Remarks/Limitations
asinh	MATLAB	—
atan	MATLAB	—
atan2	MATLAB	—
atand	MATLAB	—
atanh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
beta	MATLAB	—
betainc	MATLAB	—
betaln	MATLAB	—
bitand	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitand	Fixed-Point Toolbox	—
bitcmp	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitcmp	Fixed-Point Toolbox	—
bitget	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitget	Fixed-Point Toolbox	—
bitor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.

Function	Product	Remarks/Limitations
bitor	Fixed-Point Toolbox	—
bitrevorder	Signal Processing Toolbox	<ul style="list-style-type: none"> Requires Signal Processing Blockset license.
bitset	MATLAB	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitshift	Fixed-Point Toolbox	—
bitxor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitxor	Fixed-Point Toolbox	—
cart2pol	MATLAB	—
cart2sph	MATLAB	—
cast	MATLAB	—
cat	MATLAB	<ul style="list-style-type: none"> Accepts up to three input arguments.
ceil	MATLAB	—
char	MATLAB	—
chol	MATLAB	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> • If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> ▪ 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.
disp	Fixed-Point Toolbox	—
divide	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Complex and imaginary divisors are not supported.
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
eig	MATLAB	<ul style="list-style-type: none"> 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: <div>[V,D] = eig(A,eye(size(A)),'qz')</div> <p>However, V may represent a different basis of eigenvectors, and the eigenvalues in D may not be in the same order.</p> Options 'balance', 'nobalance', and 'chol' are not yet supported. Outputs are always of complex type.
ellipke	MATLAB	—
end	Fixed-Point Toolbox	—
eps	MATLAB	—
eps	Fixed-Point Toolbox	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only.
eq	MATLAB	—
eq	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—
exp	MATLAB	—
expint	MATLAB	—
expm	MATLAB	—

Function	Product	Remarks/Limitations
expm1	MATLAB	—
eye	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
factorial	MATLAB	—
false	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
fft	MATLAB	<ul style="list-style-type: none"> • Length of input vector must be a power of 2.
fftshift	MATLAB	—
fi	Fixed-Point Toolbox	<ul style="list-style-type: none"> • 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 <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v,'PropertyName',PropertyValue...)</code>. • Works for constant input values only; that is, the value of the input must be known at compile time. • <code>numerictype</code> object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter2	MATLAB	—

Function	Product	Remarks/Limitations
fimath	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
get	Fixed-Point Toolbox	<ul style="list-style-type: none"> The syntax structure = get(o) is not supported.
gt	MATLAB	—
gt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
hilb	MATLAB	—
histc	MATLAB	—
horzcat	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
hypot	MATLAB	—
idivide	MATLAB	<ul style="list-style-type: none"> • opt string must be in lowercase. • For efficient generated code, MATLAB divide-by-zero rules are supported only for the 'round' option.
ifft	MATLAB	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • No support for N-dimensional matrices. Size vector must have exactly two elements.
inf	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
int8, int16, int32	MATLAB	—
int8, int16, int32	Fixed-Point Toolbox	—
interp1	MATLAB	<ul style="list-style-type: none"> • 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.
interp1q, see interp1	MATLAB	<ul style="list-style-type: none"> • X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.
intmax	MATLAB	
intmin	MATLAB	

Function	Product	Remarks/Limitations
inv	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	—
islogical	MATLAB	—
isnan	MATLAB	—
isnan	Fixed-Point Toolbox	—
isnumeric	MATLAB	—

Function	Product	Remarks/Limitations
isnumeric	Fixed-Point Toolbox	—
isnumerictype	Fixed-Point Toolbox	—
isreal	MATLAB	—
isreal	Fixed-Point Toolbox	—
isrow	Fixed-Point Toolbox	—
isscalar	MATLAB	—
isscalar	Fixed-Point Toolbox	—
assigned	Fixed-Point Toolbox	—
issorted	MATLAB	—
isstruct	MATLAB	—
isvector	MATLAB	—
isvector	Fixed-Point Toolbox	—
kron	MATLAB	
lcm	MATLAB	—
ldivide	MATLAB	—
le	MATLAB	—
le	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
linspace	MATLAB	<ul style="list-style-type: none"> Number of points N must be a constant that is positive, real, and integer valued
log	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Toolbox	—
logspace	MATLAB	—
lowerbound	Fixed-Point Toolbox	—
lsb	Fixed-Point Toolbox	—
lt	MATLAB	—
lt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
lu	MATLAB	—
magic	MATLAB	—
max	MATLAB	—
max	Fixed-Point Toolbox	—
mean	MATLAB	—
median	MATLAB	—
meshgrid	MATLAB	—

Function	Product	Remarks/Limitations
min	MATLAB	—
min	Fixed-Point Toolbox	—
minus	MATLAB	—
minus	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Does not support third output argument C (cell array)
mpower	MATLAB	—
mrdivide	MATLAB	—
mtimes	MATLAB	—
mtimes	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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.
NaN or nan	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants Supports only one or two dimension arguments
nargin	MATLAB	—
nargout	MATLAB	—
nchoosek	MATLAB	—

Function	Product	Remarks/Limitations
ndims	MATLAB	—
ndims	Fixed-Point Toolbox	—
ne	MATLAB	—
ne	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
nextpow2	MATLAB	—
norm	MATLAB	—
normest	MATLAB	—
not	MATLAB	—
nthroot	MATLAB	—
numberofelements	Fixed-Point Toolbox	<ul style="list-style-type: none"> numberofelements and numel both work the same as MATLAB numel for fi objects in Embedded MATLAB.
numerictype	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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.
ones	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, non-negative, integer constants
or	MATLAB	—
pascal	MATLAB	—
permute	MATLAB	—
permute	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
pi	MATLAB	—
pinv	MATLAB	—
planerot	MATLAB	—
plus	MATLAB	—
plus	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Does not discard non-finite input values Complex input always produces complex output
polyfit	MATLAB	—
polyval	MATLAB	—
pow2	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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.
power	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 RTW 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	—

Function	Product	Remarks/Limitations
rand	MATLAB	<ul style="list-style-type: none"> Does not support the V5 generator. May not match MATLAB if seeded with negative values.
randn	MATLAB	<ul style="list-style-type: none"> 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	—
realmin	MATLAB	—
realmin	Fixed-Point Toolbox	—
realpow	MATLAB	—
realsqrt	MATLAB	—
rem	MATLAB	—
repmat	MATLAB	—
repmat	Fixed-Point Toolbox	—
rescale	Fixed-Point Toolbox	—
reshape	MATLAB	<ul style="list-style-type: none"> Accepts a maximum of three arguments

Function	Product	Remarks/Limitations
reshape	Fixed-Point Toolbox	—
rot90	MATLAB	—
round	MATLAB	—
sec	MATLAB	—
secd	MATLAB	—
sech	MATLAB	—
shiftdim	MATLAB	<ul style="list-style-type: none"> • 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	—
single	Fixed-Point Toolbox	—
sinh	MATLAB	—
size	MATLAB	—
size	Fixed-Point Toolbox	—
sort	MATLAB	—
sortrows	MATLAB	—
sosfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Requires Signal Processing Blockset license
sph2cart	MATLAB	—
squeeze	MATLAB	—

Function	Product	Remarks/Limitations
sqrt	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW targets when the input value <code>x</code> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
sqrt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
std	MATLAB	—
strcmp	MATLAB	—
struct	MATLAB	—
sub2ind	MATLAB	<ul style="list-style-type: none"> 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	—
subsref	Fixed-Point Toolbox	—
sum	MATLAB	—
sum	Fixed-Point Toolbox	—
svd	MATLAB	—
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—
times	MATLAB	—
times	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.

Function	Product	Remarks/Limitations
toeplitz	MATLAB	—
trace	MATLAB	—
trapz	MATLAB	—
tril	MATLAB	—
triu	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Toolbox	—
tril	Fixed-Point Toolbox	—
triu	Fixed-Point Toolbox	—
true	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, non-negative, integer constants
typecast	MATLAB	<ul style="list-style-type: none"> • 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	—

Function	Product	Remarks/Limitations
upperbound	Fixed-Point Toolbox	—
vander	MATLAB	—
var	MATLAB	—
vertcat	Fixed-Point Toolbox	—
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Dimensions must be real, non-negative, integer constants

Function	Product	Remarks/Limitations
abs	MATLAB	—
abs	Fixed-Point Toolbox	—
acos	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
acosd	MATLAB	—
acosh	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.

Function	Product	Remarks/Limitations
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	—
asin	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
asind	MATLAB	—
asinh	MATLAB	—
atan	MATLAB	—
atan2	MATLAB	—
atand	MATLAB	—

Function	Product	Remarks/Limitations
atanh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
beta	MATLAB	—
betainc	MATLAB	—
betaln	MATLAB	—
bitand	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitand	Fixed-Point Toolbox	—
bitcmp	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitcmp	Fixed-Point Toolbox	—
bitget	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitget	Fixed-Point Toolbox	—
bitor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitor	Fixed-Point Toolbox	—
bitrevorder	Signal Processing Toolbox	<ul style="list-style-type: none"> Requires Signal Processing Blockset license.

Function	Product	Remarks/Limitations
bitset	MATLAB	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitshift	Fixed-Point Toolbox	—
bitxor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitxor	Fixed-Point Toolbox	—
cart2pol	MATLAB	—
cart2sph	MATLAB	—
cast	MATLAB	—
cat	MATLAB	<ul style="list-style-type: none"> Accepts up to three input arguments.
ceil	MATLAB	—
char	MATLAB	—
chol	MATLAB	<ul style="list-style-type: none"> Does not allow two output arguments.
class	MATLAB	—
compan	MATLAB	—
complex	MATLAB	—
complex	Fixed-Point Toolbox	—
cond	MATLAB	—
conj	MATLAB	—

Function	Product	Remarks/Limitations
conj	Fixed-Point Toolbox	—
conv	MATLAB	—
conv2	MATLAB	—
cos	MATLAB	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	—
coth	MATLAB	—
cov	MATLAB	—
cross	MATLAB	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.
disp	Fixed-Point Toolbox	—
divide	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Complex and imaginary divisors are not supported.
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
eig	MATLAB	<ul style="list-style-type: none"> 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: $[V,D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')$ <p>However, V may represent a different basis of eigenvectors, and the eigenvalues in D may not be in the same order.</p> Options 'balance', 'nobalance', and 'chol' are not yet supported. Outputs are always of complex type.
ellipke	MATLAB	—
end	Fixed-Point Toolbox	—
eps	MATLAB	—
eps	Fixed-Point Toolbox	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only.
eq	MATLAB	—
eq	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—
exp	MATLAB	—
expint	MATLAB	—
expm	MATLAB	—

Function	Product	Remarks/Limitations
expm1	MATLAB	—
eye	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
factorial	MATLAB	—
false	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
fft	MATLAB	<ul style="list-style-type: none"> • Length of input vector must be a power of 2.
fftshift	MATLAB	—
fi	Fixed-Point Toolbox	<ul style="list-style-type: none"> • 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 <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v,'PropertyName',PropertyValue...)</code>. • Works for constant input values only; that is, the value of the input must be known at compile time. • <code>numerictype</code> object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter2	MATLAB	—

Function	Product	Remarks/Limitations
fimath	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
get	Fixed-Point Toolbox	<ul style="list-style-type: none"> The syntax structure = get(o) is not supported.
gt	MATLAB	—
gt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
hilb	MATLAB	—
histc	MATLAB	—
horzcat	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
hypot	MATLAB	—
idivide	MATLAB	<ul style="list-style-type: none"> • opt string must be in lowercase. • For efficient generated code, MATLAB divide-by-zero rules are supported only for the 'round' option.
ifft	MATLAB	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • No support for N-dimensional matrices. Size vector must have exactly two elements.
inf	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
int8, int16, int32	MATLAB	—
int8, int16, int32	Fixed-Point Toolbox	—
interp1	MATLAB	<ul style="list-style-type: none"> • 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.
interp1q, see interp1	MATLAB	<ul style="list-style-type: none"> • X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.
intmax	MATLAB	
intmin	MATLAB	

Function	Product	Remarks/Limitations
inv	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	—
islogical	MATLAB	—
isnan	MATLAB	—
isnan	Fixed-Point Toolbox	—
isnumeric	MATLAB	—

Function	Product	Remarks/Limitations
isnumeric	Fixed-Point Toolbox	—
isnumerictype	Fixed-Point Toolbox	—
isreal	MATLAB	—
isreal	Fixed-Point Toolbox	—
isrow	Fixed-Point Toolbox	—
isscalar	MATLAB	—
isscalar	Fixed-Point Toolbox	—
assigned	Fixed-Point Toolbox	—
issorted	MATLAB	—
isstruct	MATLAB	—
isvector	MATLAB	—
isvector	Fixed-Point Toolbox	—
kron	MATLAB	—
lcm	MATLAB	—
ldivide	MATLAB	—
le	MATLAB	—
le	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
linspace	MATLAB	<ul style="list-style-type: none"> Number of points N must be a constant that is positive, real, and integer valued
log	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Toolbox	—
logspace	MATLAB	—
lowerbound	Fixed-Point Toolbox	—
lsb	Fixed-Point Toolbox	—
lt	MATLAB	—
lt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
lu	MATLAB	—
magic	MATLAB	—
max	MATLAB	—
max	Fixed-Point Toolbox	—
mean	MATLAB	—
median	MATLAB	—
meshgrid	MATLAB	—

Function	Product	Remarks/Limitations
<code>min</code>	MATLAB	—
<code>min</code>	Fixed-Point Toolbox	—
<code>minus</code>	MATLAB	—
<code>minus</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
<code>mldivide</code>	MATLAB	—
<code>mod</code>	MATLAB	—
<code>mode</code>	MATLAB	<ul style="list-style-type: none"> Does not support third output argument <code>C</code> (cell array)
<code>mpower</code>	MATLAB	—
<code>mrdivide</code>	MATLAB	—
<code>mtimes</code>	MATLAB	—
<code>mtimes</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
<code>NaN</code> or <code>nan</code>	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants. Supports only one or two dimension arguments.
<code>nargin</code>	MATLAB	—
<code>nargout</code>	MATLAB	—
<code>nchoosek</code>	MATLAB	—
<code>ndims</code>	MATLAB	—
<code>ndims</code>	Fixed-Point Toolbox	—
<code>ne</code>	MATLAB	—
<code>ne</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.

Function	Product	Remarks/Limitations
nextpow2	MATLAB	—
norm	MATLAB	—
normest	MATLAB	—
not	MATLAB	—
nthroot	MATLAB	—
numberofelements	Fixed-Point Toolbox	<ul style="list-style-type: none"> numberofelements and numel both work the same as MATLAB numel for fi objects in Embedded MATLAB.
numerictype	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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 non-fixed-point signal. Use to create numerictype objects in Embedded MATLAB code.
ones	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants.
or	MATLAB	—
pascal	MATLAB	—
permute	MATLAB	—
permute	Fixed-Point Toolbox	—
pi	MATLAB	—
pinv	MATLAB	—
planerot	MATLAB	—
plus	MATLAB	—

Function	Product	Remarks/Limitations
plus	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
pol2cart	MATLAB	—
poly	MATLAB	<ul style="list-style-type: none"> Does not discard nonfinite input values. Complex input always produces complex output.
polyfit	MATLAB	—
polyval	MATLAB	—
pow2	Fixed-Point Toolbox	<ul style="list-style-type: none"> For the syntax <code>pow2(a, K)</code>, <code>K</code> must be a constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
power	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN for RTW targets when both <code>X</code> and <code>Y</code> are real, but <code>power(X,Y)</code> is complex. To get the complex result, make the input value <code>X</code> complex by passing in <code>complex(X)</code>. For example, <code>power(complex(X),Y)</code>. Generates an error during simulation and returns NaN for RTW targets when both <code>X</code> and <code>Y</code> are real, but <code>X.^Y</code> is complex. To get the complex result, make the input value <code>X</code> complex by using <code>complex(X)</code>. For example, <code>complex(X).^Y</code>.
prod	MATLAB	—
qr	MATLAB	—
rand	MATLAB	<ul style="list-style-type: none"> Does not support the V5 generator. May not match MATLAB if seeded with negative values.
randn	MATLAB	<ul style="list-style-type: none"> May not match MATLAB if seeded with negative values.

Function	Product	Remarks/Limitations
range	Fixed-Point Toolbox	—
rank	MATLAB	—
rcond	MATLAB	—
rdivide	MATLAB	—
real	MATLAB	—
real	Fixed-Point Toolbox	—
reallog	MATLAB	—
realmax	MATLAB	—
realmax	Fixed-Point Toolbox	—
realmin	MATLAB	—
realmin	Fixed-Point Toolbox	—
realpow	MATLAB	—
realsqrt	MATLAB	—
rem	MATLAB	—
repmat	MATLAB	—
repmat	Fixed-Point Toolbox	—
rescale	Fixed-Point Toolbox	—
reshape	MATLAB	<ul style="list-style-type: none"> Accepts a maximum of three arguments.
reshape	Fixed-Point Toolbox	—
rot90	MATLAB	—
round	MATLAB	—

Function	Product	Remarks/Limitations
sec	MATLAB	—
secd	MATLAB	—
sech	MATLAB	—
shiftdim	MATLAB	<ul style="list-style-type: none"> • 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	—
single	Fixed-Point Toolbox	—
sinh	MATLAB	—
size	MATLAB	—
size	Fixed-Point Toolbox	—
sort	MATLAB	—
sortrows	MATLAB	—
sosfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Requires Signal Processing Blockset license
sph2cart	MATLAB	—
squeeze	MATLAB	—
sqrt	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN for RTW 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 <code>complex(x)</code>.

Function	Product	Remarks/Limitations
sqrt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
std	MATLAB	—
strcmp	MATLAB	—
struct	MATLAB	—
sub2ind	MATLAB	<ul style="list-style-type: none"> 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	—
subsref	Fixed-Point Toolbox	—
sum	MATLAB	—
sum	Fixed-Point Toolbox	—
svd	MATLAB	—
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—
times	MATLAB	—
times	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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	—
tril	MATLAB	—

Function	Product	Remarks/Limitations
triu	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Toolbox	—
tril	Fixed-Point Toolbox	—
triu	Fixed-Point Toolbox	—
true	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
typecast	MATLAB	<ul style="list-style-type: none"> • 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	—

Function	Product	Remarks/Limitations
vertcat	Fixed-Point Toolbox	—
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Dimensions must be real, nonnegative, integer constants.

Embedded MATLAB Run-Time Library – Categorical List

The following topics list functions in the Embedded MATLAB run-time 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.

- “Arithmetic Operator Functions” on page 1-59
- “Casting Functions” on page 1-60
- “Complex Number Functions” on page 1-61
- “Derivative and Integral Functions” on page 1-61
- “Discrete Math Functions” on page 1-61
- “Exponential Functions” on page 1-62
- “Filtering and Convolution Functions” on page 1-62
- “Fixed-Point Toolbox Functions” on page 1-63
- “Histogram Functions” on page 1-66
- “Input and Output Functions” on page 1-66
- “Interpolation and Computational Geometry” on page 1-67

- “Logical Operator Functions” on page 1-67
- “Matrix and Array Functions” on page 1-68
- “Polynomial Functions” on page 1-71
- “Relational Operator Functions” on page 1-71
- “Rounding and Remainder Functions” on page 1-72
- “Set Functions” on page 1-72
- “Signal Processing Functions” on page 1-72
- “Special Values” on page 1-73
- “Specialized Math” on page 1-73
- “Statistical Functions” on page 1-75
- “String Functions” on page 1-75
- “Structure Functions” on page 1-75
- “Trigonometric Functions” on page 1-76

For an alphabetical list of these functions, and remarks and limitations for them, see “Embedded MATLAB Run-Time Function Library — Alphabetical List” on page 1-18.

Arithmetic Operator Functions

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

Function	Description
<code>ctranspose</code>	Complex conjugate transpose (')
<code>idivide</code>	Integer division with rounding option
<code>isa</code>	Determine if input is object of given class
<code>ldivide</code>	Left array divide
<code>minus</code>	Minus (-)
<code>mldivide</code>	Left matrix divide (\)

Function	Description
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
log1p	Compute $\log(1+x)$ accurately for small values of x
nextpow2	Next higher power of 2
nthroot	Real n th 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

Function	Description
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 <code>fi</code> object
all	Determine whether all array elements are nonzero
any	Determine whether any array elements are nonzero
bitand	Bitwise AND of two <code>fi</code> objects
bitcmp	Bitwise complement of <code>fi</code> object
bitget	Bit at certain position
bitor	Bitwise OR of two <code>fi</code> objects
bitset	Set bit at certain position
bitshift	Shift bits specified number of places
bitxor	Bitwise exclusive OR of two <code>fi</code> objects
complex	Construct complex <code>fi</code> object from real and imaginary parts
conj	Complex conjugate of <code>fi</code> object
ctranspose	Complex conjugate transpose of <code>fi</code> object
diag	Diagonal matrices or diagonals of matrix
disp	Display object
divide	Divide two objects
double	Double-precision floating-point real-world value of <code>fi</code> 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
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
issigned	Determine whether fi object is signed
isvector	Determine whether input is vector

Function	Description
le	Determine whether real-world value of <code>fi</code> object is less than or equal to another
length	Vector length
logical	Convert numeric values to logical
lowerbound	Lower bound of range of <code>fi</code> object
lsb	Scaling of least significant bit of <code>fi</code> object
lt	Determine whether real-world value of one <code>fi</code> object is less than another
max	Largest element in array of <code>fi</code> objects
min	Smallest element in array of <code>fi</code> objects
minus	Matrix difference between <code>fi</code> objects
mtimes	Matrix product of <code>fi</code> objects
ndims	Number of array dimensions
ne	Determine whether real-world values of two <code>fi</code> objects are not equal
numberofelements	Number of data elements in <code>fi</code> array
numerictype	Construct <code>numerictype</code> object
permute	Rearrange dimensions of multidimensional array
plus	Matrix sum of <code>fi</code> objects
pow2	Multiply by 2^k
range	Numerical range of <code>fi</code> 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 <code>fi</code> object
reshape	Reshape array
sign	Perform signum function on array

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

Function	Description
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 (simulation only)
isinteger	Determine if input is integer array
islogical	Determine if input is logical array
isnan	Detect NaN elements of an array (simulation only)
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
max	Maximum elements of a matrix

Function	Description
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
toeplitz	Toeplitz matrix

Function	Description
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

Function	Description
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
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

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

Symbol	Description
expint	Exponential integral
gamma	Gamma function
gammainc	Incomplete gamma function
gamma1n	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

Function	Description
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
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

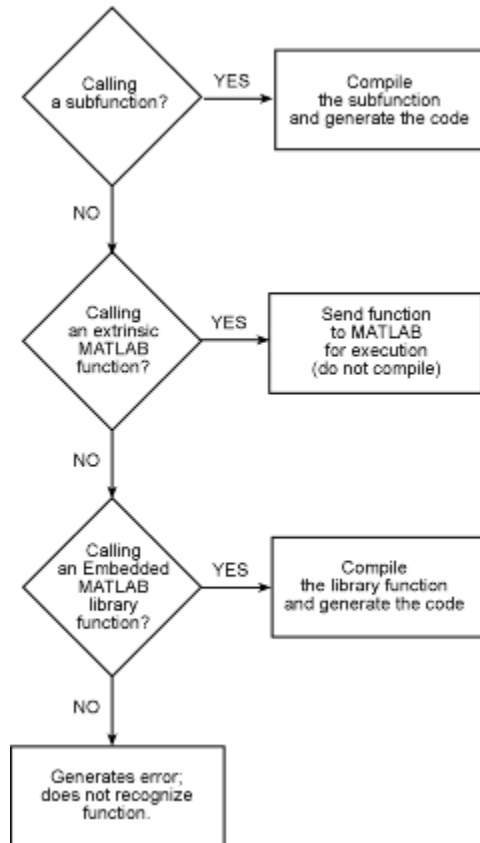
Calling Functions in Embedded MATLAB

This section describes how to call subfunctions, Embedded MATLAB run-time library functions, and MATLAB functions in Embedded MATLAB.

- “How Embedded MATLAB Resolves Function Calls” on page 1-78
- “Calling Subfunctions” on page 1-80
- “Calling Embedded MATLAB Run-Time Library Functions” on page 1-80
- “Calling MATLAB Functions” on page 1-81

How Embedded MATLAB Resolves Function Calls

During code generation for simulation targets, Embedded MATLAB attempts to resolve function calls as follows:

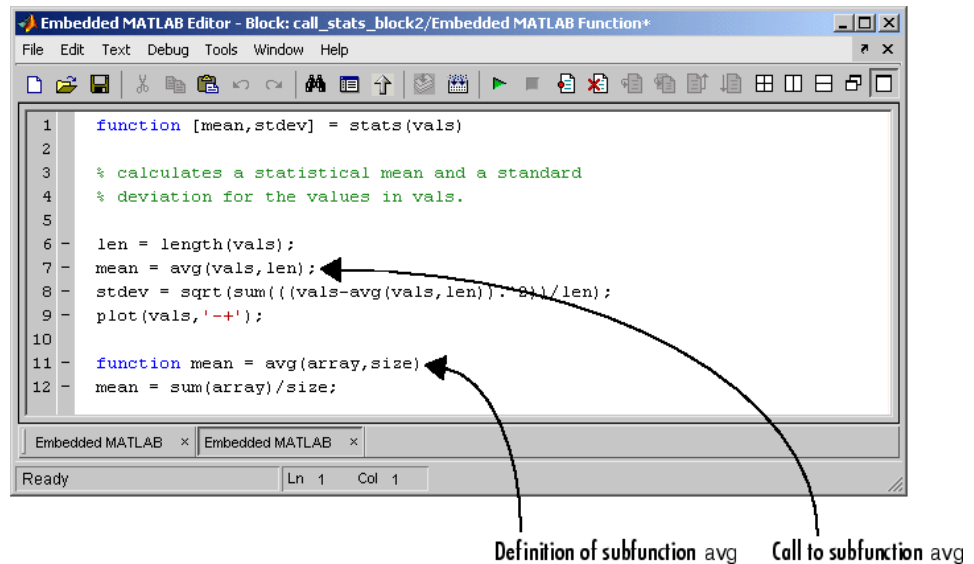


Embedded MATLAB functions attempt to resolve function calls first as subfunctions, then as extrinsic functions on the MATLAB path, and finally as Embedded MATLAB run-time library functions. Each type of function has its own requirements and behavior in Embedded MATLAB. For example, you must declare MATLAB functions to be *extrinsic* before calling them from an Embedded MATLAB function (see “Calling MATLAB Functions” on page 1-81).

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 Run-Time Library Functions

The Embedded MATLAB run-time library is a subset of MATLAB, Fixed-Point Toolbox, and Signal Processing Toolbox functions that can be used to generate code.

Supported Embedded MATLAB run-time library functions appear in “Embedded MATLAB Run-Time Function Library” on page 1-18.

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

To call MATLAB functions on the path, you must first declare them as extrinsic functions in Embedded MATLAB. 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 “Code Generation for MATLAB Function Calls” on page 1-85).

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 1-82).
- Call the MATLAB function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 1-84).

This section describes how to call MATLAB functions from Embedded MATLAB:

- “Declaring MATLAB Functions as Extrinsic Functions” on page 1-82
- “Calling MATLAB Functions Using `feval`” on page 1-84
- “Code Generation for MATLAB Function Calls” on page 1-85
- “Working with Opaque Values” on page 1-86
- “Restrictions on Extrinsic Functions in Embedded MATLAB” on page 1-88

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 “Code Generation for MATLAB Function Calls” on page 1-85).
- 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 opaque values (see “Working with Opaque Values” on page 1-86).
- 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 1-83).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 1-83). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 1-84).

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 1-84.

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.

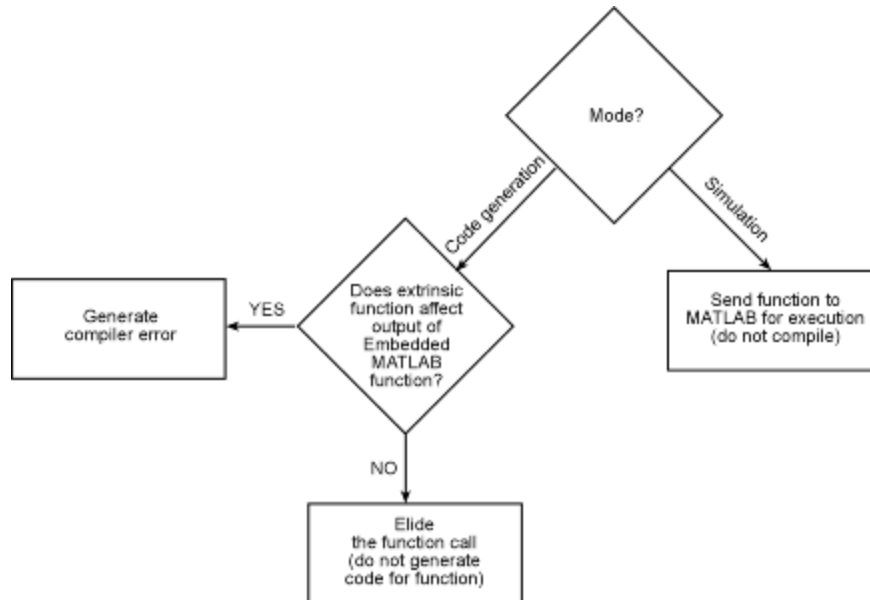
Consider the following example:

```
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`.

Code Generation for MATLAB Function Calls

Embedded MATLAB interprets extrinsic calls to MATLAB functions for code generation, 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 opaque values to an output variable (see “Working with Opaque Values” on page 1-86). 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 Opaque Values

The output of an extrinsic function is an opaque value. *Opaque values* are values of type `mxArray` — also called MATLAB type. The only valid operations for opaque values are:

- Storing opaque values in variables
- Passing opaque values to functions and returning them from functions
- Converting opaque values to nonopaque values at run time

To use values returned by extrinsic functions in other operations, you must first convert them to nonopaque values, as described in “Converting Opaque Values to Nonopaque Values” on page 1-86.

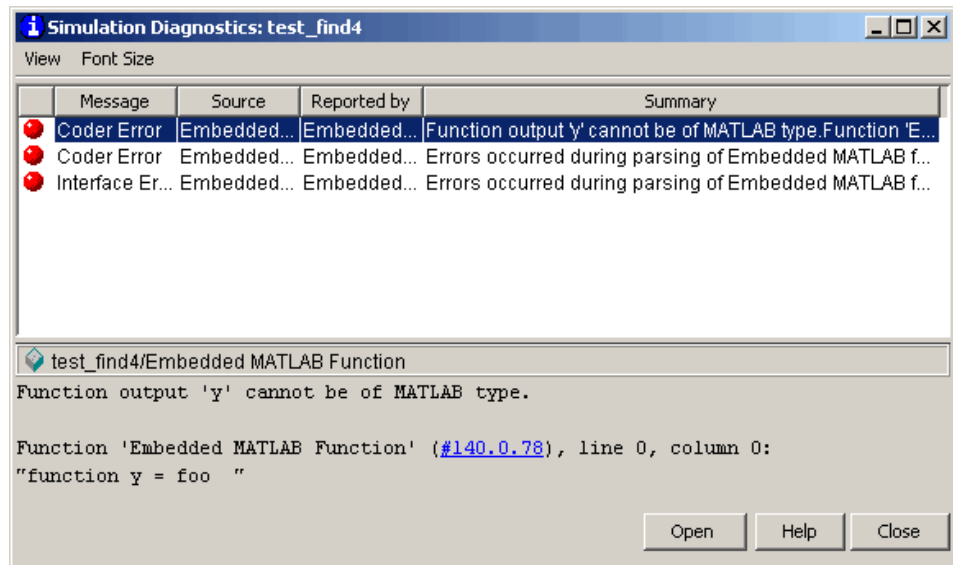
Converting Opaque Values to Nonopaque Values. To convert opaque values to nonopaque values, assign the opaque value to a variable whose type is known. At run time, Embedded MATLAB converts the opaque value to the type of the variable assigned to it. However, if the data in the opaque value is not consistent 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 opaque values representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these opaque values to another extrinsic MATLAB function — in this case, `min` — you cannot assign the opaque value 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:



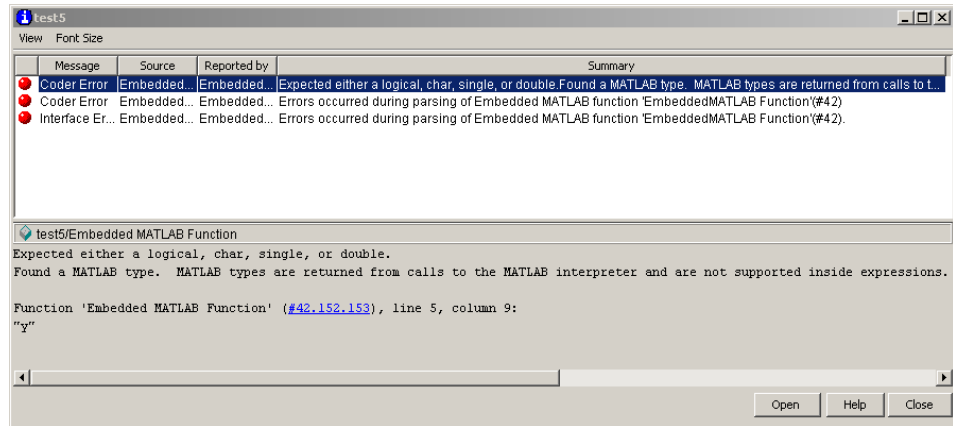
To correct this problem, declare `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
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 opaque value 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 opaque value `y` to a double array `x`:



The value `y` is opaque 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 opaque value returned by `find(x)` to a nonopaque value — 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

Using Structures

This section describes how to use MATLAB structures in Embedded MATLAB. By imposing some restrictions, Embedded MATLAB compiles MATLAB structures to generate efficient C code in Real-Time Workshop®.

- “About Embedded MATLAB Structures” on page 1-90
- “Elements of Embedded MATLAB Structures” on page 1-90
- “Types of Embedded MATLAB Structures” on page 1-91
- “Defining Local Structure Variables” on page 1-92
- “Defining Outputs as Structures” on page 1-95
- “Making Structures Persistent” on page 1-96
- “Indexing SubStructures and Fields” on page 1-96
- “Assigning Values to Structures and Fields” on page 1-97
- “Limitations with Structures” on page 1-97

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). 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 1-97).

Types of Embedded MATLAB Structures

You can define the following types of Embedded MATLAB structures:

Type	How to Define	Details
Input	<p>Depends on how you use Embedded MATLAB:</p> <ul style="list-style-type: none"> • 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”) 	<p>See:</p> <ul style="list-style-type: none"> • “Using Bus Objects” in the Simulink User’s Guide documentation • “Specifying Properties of Primary Function Inputs” on page 2-15
Output	Define structure variable in Embedded MATLAB function	See “Defining Outputs as Structures” on page 1-95.

Type	How to Define	Details
Local	Define local structure variable in Embedded MATLAB function	See “Defining Local Structure Variables” on page 1-92.
Persistent	Declare structure variable to be persistent in Embedded MATLAB function	See “Making Structures Persistent” on page 1-96.

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 1-96).

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

- “Defining Scalar Structures” on page 1-92
- “Defining Arrays of Structures” on page 1-94

Defining Scalar Structures

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

- “Defining Scalar Structures by Extension” on page 1-92
- “Defining Scalar Structures Using the MATLAB struct Function” on page 1-93
- “Defining Scalar Structures by Assignment” on page 1-93

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 1-96.

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 1-94.

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 1-97). There are several ways to create arrays of structures in Embedded MATLAB:

- “Defining an Array of Structures from a Scalar Structure” on page 1-94
- “Defining an Array of Structures Using Concatenation” on page 1-95

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 1-92.
- 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 1-92). 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 1-13.

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 1-100).

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 1-90).
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 1-98
- “Do Not Assign Empty Matrices” on page 1-99

- “Do Not Assign Opaque Values to Structures” on page 1-99
- “Do Not Add New Fields After First Use of Structures” on page 1-99
- “Make Structures Uniform in Arrays” on page 1-100
- “Do Not Reference Fields Dynamically” on page 1-100
- “Do Not Use Field Values as Constants” on page 1-100
- “Reference Field Values Individually from Structure Arrays” on page 1-100

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 Opaque Values to Structures

You cannot assign opaque values to structure elements in Embedded MATLAB; you must first convert them to nonopaque values (see “Working with Opaque Values” on page 1-86).

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 1-92 for more information.

Using 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 Run-Time Function Library” on page 1-18)

Note You cannot define handles that reference extrinsic MATLAB functions (see “Calling MATLAB Functions” on page 1-81).

- Define function handles as scalar values
- Pass function handles as arguments to other functions on the Embedded MATLAB path

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 1-104

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');

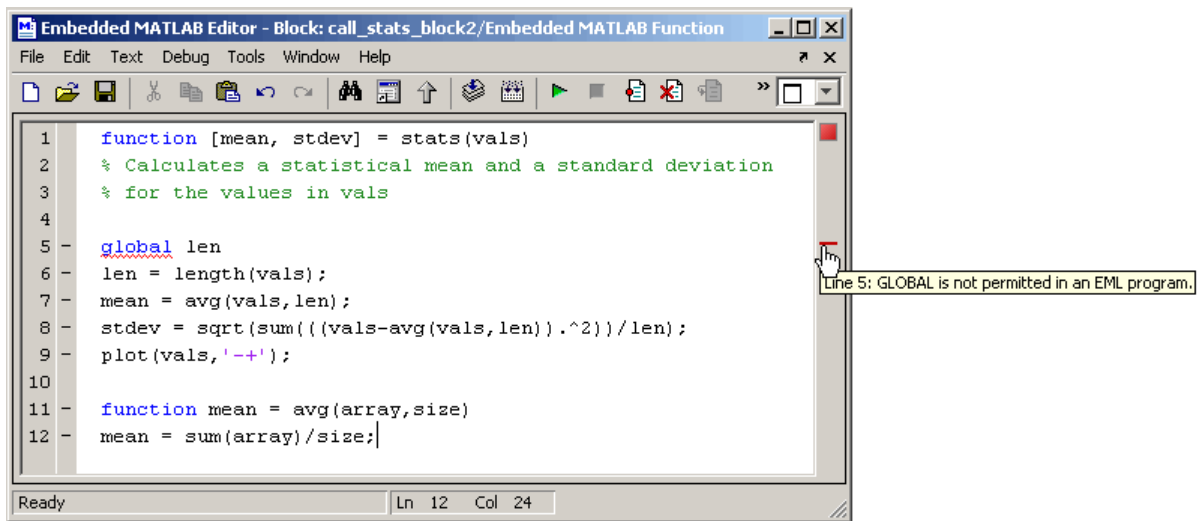
plot(data, fhandle(data));
x = fhandle(data);
```

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 (p. 2-2)	Describes what you can do with Embedded MATLAB MEX
Workflow for Converting M-Code to a C-MEX Function (p. 2-4)	Describes the steps required for preparing M-code and converting it to a C-MEX function
Installing Embedded MATLAB MEX (p. 2-5)	Explains how to install the products required for using Embedded MATLAB MEX
Setting Up the C Compiler (p. 2-6)	Explains how to install and set up a C compiler for use with Embedded MATLAB MEX
Setting Up File Infrastructure and Paths (p. 2-7)	Explains how to set up source file directories and paths
Verifying M-Code Compliance with Embedded MATLAB (p. 2-10)	Presents code verification methods and the trade-offs between two strategies: bottom-up and top-down
Specifying Properties of Primary Function Inputs (p. 2-15)	Describes how to specify the types of primary function inputs
Compiling Your M-File (p. 2-30)	Describes how to run Embedded MATLAB MEX
Working with Compilation Reports (p. 2-32)	Explains how to generate and interpret compilation reports from Embedded MATLAB MEX

About Embedded MATLAB MEX

Embedded MATLAB MEX is a compiler 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: <ul style="list-style-type: none">• 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:

```
demons
```

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.

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	Use <code>eml.extrinsic</code> or <code>feval</code> to call MATLAB functions that are not supported by Embedded MATLAB.	See “Calling MATLAB Functions” on page 1-81.
5	Make your M-code compliant with Embedded MATLAB.	See “Verifying M-Code Compliance with Embedded MATLAB ” on page 2-10.
6	Specify properties of primary function inputs.	See “Specifying Properties of Primary Function Inputs” on page 2-15.
7	Run Embedded MATLAB MEX with the appropriate command-line options.	See “Compiling Your M-File” on page 2-30.

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.

MATLAB comes with a C compiler for Windows. If you do not want to use the default C compiler or you are running on a non-Windows platform, you can 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

Embedded MATLAB MEX requires that your M-file use only the subset of language functions and features described in Chapter 1, “Working with Embedded MATLAB”. Embedded MATLAB MEX compiles M-code from M-files located in:

- The current directory
- The Embedded MATLAB path
- Embedded MATLAB libraries

You may either code your M-file from scratch or modify an existing M-file to comply with Embedded MATLAB. Depending on your approach, you can set up your file infrastructure and path to conditionally execute full-featured M-files separately from M-files that comply with Embedded MATLAB.

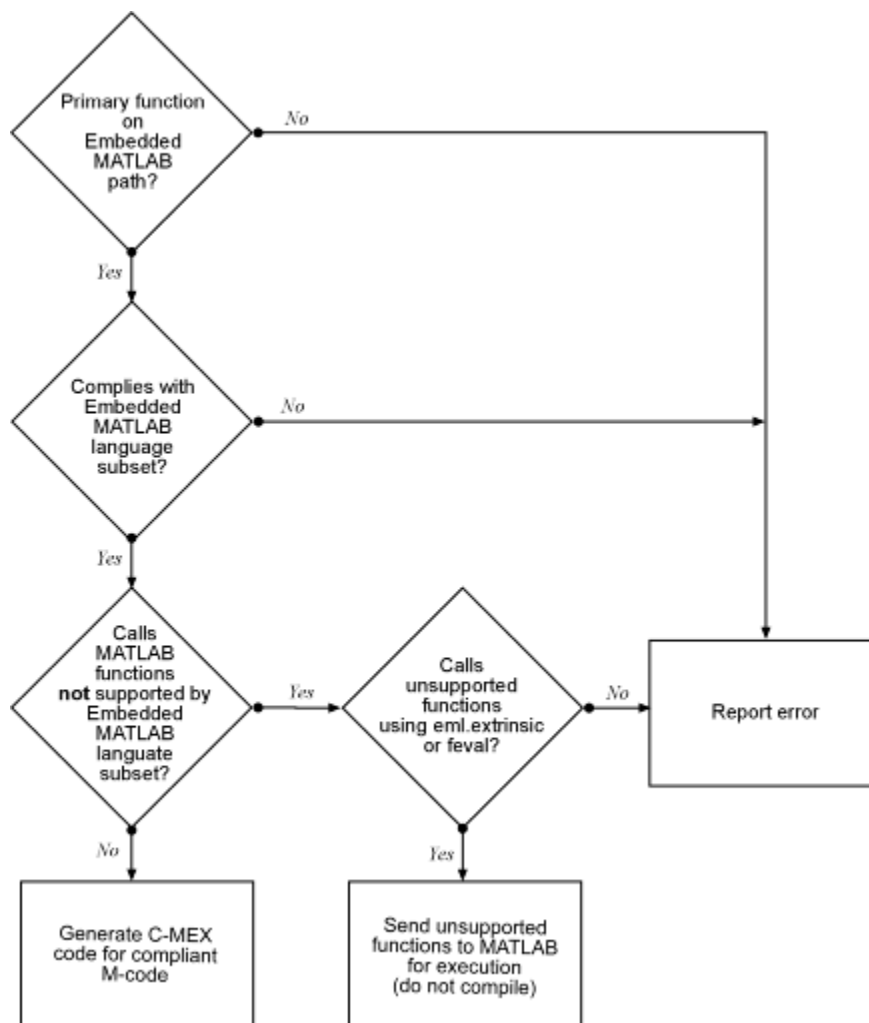
- “Embedded MATLAB Path” on page 2-7
- “Maintaining Two Versions of M-Code” on page 2-9
- “Maintaining One Version of M-Code” on page 2-9

Embedded MATLAB Path

The Embedded MATLAB path is the path where Embedded MATLAB MEX searches for M-files to convert to C-MEX functions. By default, the Embedded MATLAB path contains the current directory and the Embedded MATLAB libraries.

Embedded MATLAB MEX provides a compiler option `-I` that lets you add other directories to the Embedded MATLAB path (see `emlmex` in Chapter 3, “Functions — Alphabetical List”).

Embedded MATLAB MEX searches the Embedded MATLAB path to find your source M-files and processes them as follows:



The diagram summarizes the requirements for successfully translating M-code to C-MEX code using Embedded MATLAB MEX:

- Your primary M-function must be on the Embedded MATLAB path.
- Your primary M-function must comply with Embedded MATLAB, as described in Chapter 1, “Working with Embedded MATLAB”.

- Your primary M-function must use `eml.extrinsic` or `feval` to call MATLAB functions that are not supported by Embedded MATLAB, as described in “Calling MATLAB Functions” on page 1-81. This requirement applies to all functions called recursively from the primary function.

Maintaining Two Versions of M-Code

In this scenario, you want to preserve your full-featured M-code to run inside MATLAB, but also compile a version that complies with Embedded MATLAB to produce a C-MEX function. Follow these guidelines:

- Store the full-featured M-file in a separate directory from the Embedded MATLAB M-file.
- Add the directory containing the full-featured M-file to the MATLAB path.
- Add the directory containing the Embedded MATLAB M-file to the Embedded MATLAB path.

Note For information about how to add files to the MATLAB path, see “MATLAB Path” in the MATLAB Programming documentation.

Maintaining One Version of M-Code

In this scenario, you use the same Embedded MATLAB M-code for execution inside MATLAB as for compilation by Embedded MATLAB MEX. You do not need to preserve your full-featured M-code. Follow these guidelines:

- Store the Embedded MATLAB M-file in a dual-use directory.
- Add this directory to the MATLAB path and to the Embedded MATLAB path.

Note For information about how to add files to the MATLAB path, see “MATLAB Path” in the MATLAB Programming documentation.

Verifying M-Code Compliance with Embedded MATLAB

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

- “Code Verification Methods” on page 2-10
- “Debugging Strategies” on page 2-12
- “Reserved Prefixes” on page 2-13

Code Verification Methods

There are two methods for automatically detecting code that does not comply with Embedded MATLAB:

- Generate compilation reports that include Embedded MATLAB syntax violations using Embedded MATLAB MEX, as described in “Detecting Syntax Violations at Compile Time” on page 2-10.
- Use the MATLAB M-Lint Code Analyzer to check for Embedded MATLAB syntax violations as you write your code, as described in “Detecting Syntax Violations at Design Time” on page 2-10.

Detecting Syntax Violations at Compile Time

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 report that describes the issues and provides links to the offending M-code. See “Working with Compilation Reports” on page 2-32.

Detecting Syntax Violations at Design Time

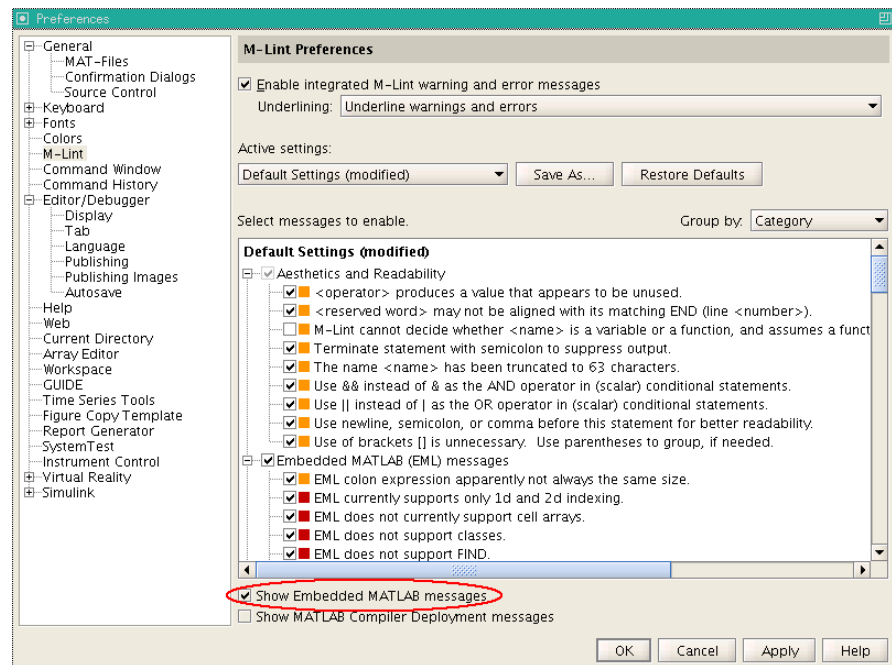
You can use the MATLAB M-Lint Code Analyzer to check M-code for common violations of Embedded MATLAB syntax as you write your code. Follow these steps:

- 1 Configure M-Lint to detect and report Embedded MATLAB syntax violations, as follows:

- a In the MATLAB desktop, select **File > Preferences** to open the Preferences dialog.
- b In the left pane of the Preferences dialog, select **M-Lint**.

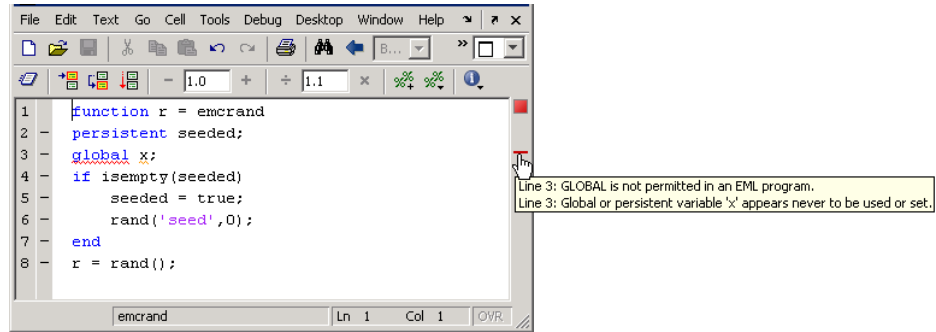
The M-Lint Preferences dialog appears in the right pane, with Embedded MATLAB messages enabled.

- c At the bottom of the M-Lint Preferences dialog, select the check box **Show Embedded MATLAB messages**, as shown:



Note M-Lint checks for only the syntax violations that you select under the **Embedded MATLAB (EML) messages** check box in the Default Settings pane of the M-Lint Preferences dialog box. To check for *all* possible errors, see “Detecting Syntax Violations at Compile Time” on page 2-10.

- 2 Use M-Lint to generate reports or display messages in the MATLAB Editor/Debugger, as in this example:



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

Debugging Strategies

Before using these verification tools, 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	<ol style="list-style-type: none"> 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. 	<ul style="list-style-type: none"> • Efficient • Safe • Easy to isolate Embedded MATLAB syntax violations 	Requires application tests that work from the bottom up
Top-down verification	<ol style="list-style-type: none"> 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 1-82). 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	<p>Introduces extraneous code that you must remove after code verification, including:</p> <ul style="list-style-type: none"> • Extrinsic declarations • Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values (see “Working with Opaque Values” on page 1-86)

Reserved Prefixes

Embedded MATLAB reserves the prefixes `eml` and `eml_` for global C functions and variables in generated code. For example, Embedded MATLAB run-time 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` or `eml_`.

Specifying Properties of Primary Function Inputs

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 or *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, that is, functions called by the primary function.

- “Properties to Specify” on page 2-15
- “Rules for Specifying Properties of Primary Inputs” on page 2-17
- “Methods for Defining Properties of Primary Inputs” on page 2-17
- “Defining Input Properties by Example at the Command Line” on page 2-18
- “Defining Input Properties Programmatically in the M-File” on page 2-21

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	✓	✓	✓	✓	✓
All other inputs	✓	✓	✓		

Note When a primary input is a structure, Embedded MATLAB MEX treats each field as a separate input. Therefore, you must specify properties for each field of a primary structure input, as follows:

- For all fields of input structures, specify class, size, and complexity.
 - For fields that are fixed-point class, also specify `numerictype`, and `fimath`.
-

Default Property Values

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

Property	Default
class	No default Caution You must explicitly specify the class of all primary function inputs.
size	scalar
complexity	real
numerictype	No default
fimath	No default

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

Class Name	Description
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 number array
double	Double-precision floating-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:

- Specify the class of all primary function inputs because Embedded MATLAB MEX does not assign a default class.
- 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 class, type, and complexity of each of its fields. For every field whose class is `fi`, you must also specify the `numerictype` and `fimath` properties.

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 2-18	<ul style="list-style-type: none">• Easy to use• Does not alter original M-code• Designed for prototyping a function that has a small number of primary inputs	<ul style="list-style-type: none">• 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 2-21	<ul style="list-style-type: none">• 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.

- “Syntax of `emlmex -eg`” on page 2-19
- “Rules for using the `-eg` option” on page 2-19
- “Examples: Specifying Properties of Primary Inputs by Example” on page 2-19
- “Examples: Specifying Properties of Primary Fixed-Point Inputs by Example” on page 2-20

Syntax of `emlmex -eg`

See `emlmex` in Chapter 3, “Functions — Alphabetical List”.

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 scalar integer and input *v* is a real 1-by-4 vector:

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

- 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 2-21
- “Rules for Using `assert` Function” on page 2-26
- “Example: Specifying General Properties of Primary Inputs” on page 2-26
- “Example: Specifying Properties of Primary Fixed-Point Inputs” on page 2-27
- “Example: Specifying Class and Size of Scalar Structure” on page 2-28
- “Example: Specifying Class and Size of Structure Array” on page 2-29

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. 2-22)	<code>assert (isa (param, 'class_name'))</code>
Specify fi Class (p. 2-22)	<code>assert (isfi (param))</code>
Specify Structure Class (p. 2-23)	<code>assert (isstruct (param))</code>
Specify Any Size (p. 2-23)	<code>assert (all (size(param) == [dims]))</code>
Specify Scalar Size (p. 2-24)	<code>assert (isscalar((param))</code>
Specify Real Input (p. 2-24)	<code>assert (isreal((param))</code>
Specify Complex Input (p. 2-24)	<code>assert (~isreal((param))</code>
Specify numeric type of Fixed-Point Input (p. 2-25)	<code>assert (isequal (numericity(fi param), T))</code>

Specify fimath of Fixed-Point Input (p. 2-25)	<code>assert (isequal (fimath(<i>fiparam</i>),<i>F</i>))</code>
Specify Multiple Properties of Input (p. 2-25)	<code>assert (<i>function1</i>(<i>params</i>) && <i>function2</i>(<i>params</i>) && <i>function3</i>(<i>params</i>) && ...)</code>

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 2-25 and “Specify fimath of Fixed-Point Input” on page 2-25).

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.

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 2-25 and “Specify `fimath` of Fixed-Point Input” on page 2-25).

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 class, size, and complexity 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 numericity of Fixed-Point Input.

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

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

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

% Set the numericity property of input U to T.
assert(isequal(numericity(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 2-25 and “Specify `fimath` of Fixed-Point Input” on page 2-25).
- 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)

Input	Property	Value
win	class	double
	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(isa(win, 'double'));
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

Property	Value
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 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(isstruct(S));
assert(isa(S.r,'double'));

```

```
assert(isa(S.i,'int8');  
...  

```

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'));
```

Compiling Your M-File

This topic describes how to compile your M-file with Embedded MATLAB MEX. Before beginning, make sure you have followed the prerequisite steps in “Workflow for Converting M-Code to a C-MEX Function” on page 2-4.

- “Running Embedded MATLAB MEX” on page 2-30
- “Generated Files and Locations” on page 2-31
- “File Naming Conventions” on page 2-31

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:

- Looks in the current directory for the function *M_fcn* stored in the file *M_fcn.m*
- 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 “File Naming Conventions” on page 2-31.
- 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 2-31. Error reports are described in “Working with Compilation Reports” on page 2-32.
- 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 `emlmex` in Chapter 3, “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/ <i>M_fcn_name</i> / <i>M_fcn_name_report</i> .mat
HTML reports (if errors or warnings occur during compilation)	Default output directory: emcprj/mexfcn/ <i>M_fcn_name</i> /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`).

File Naming Conventions

Embedded MATLAB MEX follows MATLAB conventions by providing platform-specific extensions for generated C-MEX files, as follows:

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

Working with Compilation Reports

Embedded MATLAB MEX automatically generates reports in HTML format when errors or warnings occur at compile time. You can use these reports to help you debug your M-code and verify compliance with Embedded MATLAB.

- “Location of Compilation Reports” on page 2-32
- “Description of Compilation Reports” on page 2-32

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:

	emcsincos
	Unsupported: GLOBAL variables.
	Function 'plot' resolved in the MATLAB workspace.

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. Implicit evaluation in MATLAB is not supported. Please declare this function extrinsic using eml.extrinsic('plot'), or call it using feval.

Function emcsincos

```

1  function r = emcsincos(num)
2  global x;
3  assert(isa(num,'double'));
4  assert(all(size(num) == [1 10]));
5  r = sin(num);
6  x = cos(num);
7  ➡ plot(num,r);

```

Functions — Alphabetical List

Purpose	Generate C-MEX code from M-code
Syntax	<code>emlmex [-options] fun</code>
Description	<p><code>emlmex</code> is a MATLAB command that invokes Embedded MATLAB MEX. You issue the <code>emlmex</code> command from the MATLAB command prompt.</p> <p><code>emlmex [-options] fun</code> translates the M-file <code>fun.m</code> to a C-MEX file and generates all necessary wrapper files.</p> <p>By default, <code>emlmex</code>:</p> <ul style="list-style-type: none">• Converts the M-function <code>fun.m</code> to a C-MEX function• Generates a platform-specific MEX-file in the current directory• Stores generated files in the subdirectory <code>emcprj/mexfcn/fun/</code> <p>You can change the default behavior by specifying one or more compilation <i>options</i> as described in “Options” on page 3-2.</p>
Options	<p>You can specify one or more compilation options with each <code>emlmex</code> 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 <code>emlmex</code> options:</p> <ul style="list-style-type: none">• “-d Specify Output Directory” on page 3-3• “-eg Specify Input Properties by Example” on page 3-3• “-F Specify Default fimath” on page 3-3• “-g Compile C-MEX Function in Debug Mode” on page 3-4• “-I Add Directories to Embedded MATLAB Path” on page 3-4• “-N Specify Default Numeric Type” on page 3-4• “-o Specify Output File Name” on page 3-5• “-O Specify Compiler Optimization Option” on page 3-5

- “-? Display Help” on page 3-6

-d Specify Output Directory

`-d out_directory`

Store generated files in directory path specified by *out_directory*. If any directories on the path do not exist, 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 2-18 .

-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 2-18) or programmatically (see “Defining Input Properties Programmatically in the M-File” on page 2-21). 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 “Embedded MATLAB Path” on page 2-7.

-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 2-18) or programmatically (see “Defining Input Properties Programmatically in the M-File” on page 2-21). 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 “File Naming Conventions” on page 2-31).

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

Embedded MATLAB MEX generates the supporting C files with the same base name as the corresponding M-files, replacing the `.m` extension with `.c`.

-O Specify Compiler Optimization Option

`-O optimization_option`

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

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

-? 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 3-6.

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 “File Naming Conventions” on page 2-31.

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 M-function name as the root name for all files.

Specifying Primary Function Input Properties By Example

Currently, the M-function `emcrand` (described in “Sample M-File” on page 3-6) uses the `assert` function to specify that its input `num` is a real double scalar, as follows:

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

Note For information about using `assert` to specify input properties for `emlmex`, see “Defining Input Properties Programmatically in the M-File” on page 2-21.

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`.

C

- complex variables in Embedded MATLAB functions 1-11
- control flow statements in Embedded MATLAB 1-14

E

- Embedded MATLAB
 - arithmetic operators 1-15
 - calling MATLAB functions 1-81
 - calling MATLAB functions as extrinsic functions 1-82
 - calling MATLAB functions using feval 1-84
 - calling other functions 1-78
 - calling subfunctions 1-80
 - code generation for MATLAB function calls 1-85
 - control flow statements 1-14
 - converting opaque values to nonopaque values 1-86
 - creating local variables 1-10
 - declaring persistent variables 1-13
 - description 1-2
 - Embedded MATLAB run-time library 1-18
 - function handles 1-102
 - how it resolves function calls 1-78
 - initializing persistent variables 1-13
 - logical operators 1-16
 - MATLAB features not supported 1-3
 - operators 1-14
 - Real-Time Workshop targets, building 1-85
 - relational operators 1-16
 - supported MATLAB functions 1-80
 - using M-Lint 1-106
 - variable types 1-9
 - variables 1-9
 - variables, complex 1-11
 - working with opaque values 1-86
- Embedded MATLAB function

- signal processing functions 1-72
- Embedded MATLAB run-time library 1-18
 - alphabetical list of functions 1-18
 - casting functions 1-60
 - categorized list of functions 1-58
 - complex number functions 1-61
 - derivative and integral functions 1-61
 - discrete math functions 1-61
 - exponential functions 1-62
 - filtering and convolution functions 1-62
 - Fixed-Point Toolbox functions 1-63
 - histogram functions 1-66
 - input and output functions 1-66 to 1-67
 - logical operator functions 1-67
 - matrix/array functions 1-68
 - polynomial functions 1-71
 - relational operator functions 1-71
 - rounding and remainder functions 1-72
 - set functions 1-72
 - special value functions 1-73
 - specialized math functions 1-73
 - statistical functions 1-75
 - string functions 1-75
 - structure functions 1-75
 - trigonometric functions 1-76
- eml.extrinsic 1-82
- extrinsic functions 1-82

F

- function handles
 - in Embedded MATLAB 1-102

I

- initialization
 - persistent variables 1-13

M

- MATLAB

- features not supported in Embedded MATLAB 1-3
- MATLAB functions
 - and opaque values in Embedded MATLAB 1-86
- MATLAB functions in Embedded MATLAB 1-81

O

- opaque values
 - converting to nonopaque values 1-86
 - in Embedded MATLAB 1-86
- operators
 - arithmetic, in Embedded MATLAB 1-15
 - logical, in Embedded MATLAB 1-16
 - relational, in Embedded MATLAB 1-16

P

- persistent variables
 - declaring in Embedded MATLAB functions 1-13
 - initializing in Embedded MATLAB functions 1-13

S

- signal processing functions
 - for Embedded MATLAB function 1-72
- subfunctions in Embedded MATLAB 1-80

V

- variable types of Embedded MATLAB 1-9