

# MATLAB<sup>®</sup>

## The Language of Technical Computing

- Computation
- Visualization
- Programming

### External Interfaces Reference

*Version 7*



## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com)  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab)  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html)

Web  
Newsgroup  
Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com)  
[bugs@mathworks.com](mailto:bugs@mathworks.com)  
[doc@mathworks.com](mailto:doc@mathworks.com)  
[service@mathworks.com](mailto:service@mathworks.com)  
[info@mathworks.com](mailto: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.

### *MATLAB External Interfaces Reference*

© COPYRIGHT 1984–2006 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](http://www.mathworks.com/patents) for more information.

### Revision History

December 1996	First Printing	New for MATLAB 5 (release 8)
May 1997	Online only	Revised for MATLAB 5.1 (Release 9)
January 1998	Online Only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online Only	Revised for MATLAB 5.3 (Release 11)
September 2000	Online Only	Revised for 6.0 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
January 2003	Online only	Revised for MATLAB 6.5.1 (Release 13SP1)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)



## Functions — By Category

**1**

---

<b>MAT-File Access</b> .....	<b>1-2</b>
<b>MX Array Manipulation</b> .....	<b>1-2</b>
<b>MEX-Files</b> .....	<b>1-9</b>
<b>MATLAB Engine</b> .....	<b>1-11</b>

## Functions — Alphabetical List

**2**

---

**Index**



# Functions — By Category

---

MAT-File Access (p. 1-2)

Incorporate and use MATLAB® data in C and Fortran programs

MX Array Manipulation (p. 1-2)

Create and manipulate MATLAB arrays from C and Fortran MEX and Engine routines

MEX-Files (p. 1-9)

Perform operations in MATLAB environment from C and Fortran MEX-files

MATLAB Engine (p. 1-11)

Call MATLAB from C and Fortran programs

See also “External Interfaces” in MATLAB Function Reference for MATLAB interfaces to DLLs, Java, COM and ActiveX, DDE, Web services, and serial port devices.

## **MAT-File Access**

<code>matClose</code> (C and Fortran)	Close MAT-file
<code>matDeleteVariable</code> (C and Fortran)	Delete named mxArray from MAT-file
<code>matGetDir</code> (C and Fortran)	Directory of mxArrays in MAT-file
<code>matGetFp</code> (C)	File pointer to MAT-file
<code>matGetNextVariable</code> (C and Fortran)	Read next mxArray from MAT-file
<code>matGetNextVariableInfo</code> (C and Fortran)	Load array header information only
<code>matGetVariable</code> (C and Fortran)	Read mxArrays from MAT-files
<code>matGetVariableInfo</code> (C and Fortran)	Load array header information only
<code>matOpen</code> (C and Fortran)	Open MAT-file
<code>matPutVariable</code> (C and Fortran)	Write mxArrays to MAT-files
<code>matPutVariableAsGlobal</code> (C and Fortran)	Put mxArrays into MAT-files as originating from global workspace

## **MX Array Manipulation**

<code>mwIndex</code> (C and Fortran)	Type for index values
<code>mwPointer</code> (Fortran)	Declare appropriate pointer type for platform
<code>mwSize</code> (C and Fortran)	Type for size values
<code>mxAddField</code> (C and Fortran)	Add field to structure array
<code>mxArrayToString</code> (C)	Convert array to string
<code>mxAssert</code> (C)	Check assertion value for debugging purposes



<code>mxAssertS</code> (C)	Check assertion value without printing assertion text
<code>mxCalcSingleSubscript</code> (C and Fortran)	Offset from first element to desired element
<code>mxCalloc</code> (C and Fortran)	Allocate dynamic memory for array using MATLAB memory manager
<code>mxChar</code> (C)	Data type for string <code>mxArray</code>
<code>mxClassID</code> (C)	Integer value identifying class of <code>mxArray</code>
<code>mxClassIDFromClassName</code> (Fortran)	Identifier corresponding to class
<code>mxComplexity</code> (C)	Flag specifying whether <code>mxArray</code> has imaginary components
<code>mxCopyCharacterToPtr</code> (Fortran)	Copy character values from Fortran array to pointer array
<code>mxCopyComplex16ToPtr</code> (Fortran)	Copy <code>COMPLEX*16</code> values from Fortran array to pointer array
<code>mxCopyComplex8ToPtr</code> (Fortran)	Copy <code>COMPLEX*8</code> values from Fortran array to pointer array
<code>mxCopyInteger1ToPtr</code> (Fortran)	Copy <code>INTEGER*1</code> values from Fortran array to pointer array
<code>mxCopyInteger2ToPtr</code> (Fortran)	Copy <code>INTEGER*2</code> values from Fortran array to pointer array
<code>mxCopyInteger4ToPtr</code> (Fortran)	Copy <code>INTEGER*4</code> values from Fortran array to pointer array
<code>mxCopyPtrToCharacter</code> (Fortran)	Copy character values from pointer array to Fortran array
<code>mxCopyPtrToComplex16</code> (Fortran)	Copy <code>COMPLEX*16</code> values from pointer array to Fortran array
<code>mxCopyPtrToComplex8</code> (Fortran)	Copy <code>COMPLEX*8</code> values from pointer array to Fortran array

<code>mxCopyPtrToInteger1</code> (Fortran)	Copy INTEGER*1 values from pointer array to Fortran array
<code>mxCopyPtrToInteger2</code> (Fortran)	Copy INTEGER*2 values from pointer array to Fortran array
<code>mxCopyPtrToInteger4</code> (Fortran)	Copy INTEGER*4 values from pointer array to Fortran array
<code>mxCopyPtrToPtrArray</code> (Fortran)	Copy pointer values from pointer array to Fortran array
<code>mxCopyPtrToReal4</code> (Fortran)	Copy REAL*4 values from pointer array to Fortran array
<code>mxCopyPtrToReal8</code> (Fortran)	Copy REAL*8 values from pointer array to Fortran array
<code>mxCopyReal4ToPtr</code> (Fortran)	Copy REAL*4 values from Fortran array to pointer array
<code>mxCopyReal8ToPtr</code> (Fortran)	Copy REAL*8 values from Fortran array to pointer array
<code>mxCreateCellArray</code> (C and Fortran)	Create unpopulated N-D cell mxArray
<code>mxCreateCellMatrix</code> (C and Fortran)	Create unpopulated 2-D cell mxArray
<code>mxCreateCharArray</code> (C and Fortran)	Create unpopulated N-D string mxArray
<code>mxCreateCharMatrixFromStrings</code> (C and Fortran)	Create populated 2-D string mxArray
<code>mxCreateDoubleMatrix</code> (C and Fortran)	Create unpopulated 2-D, double-precision, floating-point mxArray
<code>mxCreateDoubleScalar</code> (C and Fortran)	Create scalar, double-precision array initialized to specified value
<code>mxCreateLogicalArray</code> (C)	Create N-D logical mxArray initialized to false
<code>mxCreateLogicalMatrix</code> (C)	Create 2-D, logical mxArray initialized to false

<code>mxCreateLogicalScalar (C)</code>	Create scalar, logical mxArray initialized to false
<code>mxCreateNumericArray (C and Fortran)</code>	Create unpopulated N-D numeric mxArray
<code>mxCreateNumericMatrix (C and Fortran)</code>	Create numeric matrix and initialize data elements to 0
<code>mxCreateSparse (C and Fortran)</code>	Create 2-D unpopulated sparse mxArray
<code>mxCreateSparseLogicalMatrix (C)</code>	Create unpopulated 2-D, sparse, logical mxArray
<code>mxCreateString (C and Fortran)</code>	Create 1-by-N string mxArray initialized to specified string
<code>mxCreateStructArray (C and Fortran)</code>	Create unpopulated N-D structure mxArray
<code>mxCreateStructMatrix (C and Fortran)</code>	Create unpopulated 2-D structure mxArray
<code>mxDestroyArray (C and Fortran)</code>	Free dynamic memory allocated by <code>mxCreate</code>
<code>mxDuplicateArray (C and Fortran)</code>	Make deep copy of array
<code>mxFree (C and Fortran)</code>	Free dynamic memory allocated by <code>mxMalloc</code> , <code>mxRealloc</code> , or <code>mxRealloc</code>
<code>mxGetCell (C and Fortran)</code>	Contents of mxArray cell
<code>mxGetChars (C)</code>	Pointer to character array data
<code>mxGetClassID (C and Fortran)</code>	Class of mxArray
<code>mxGetClassName (C and Fortran)</code>	Class of mxArray as string
<code>mxGetData (C and Fortran)</code>	Pointer to data
<code>mxGetDimensions (C and Fortran)</code>	Pointer to dimensions array
<code>mxGetElementSize (C and Fortran)</code>	Number of bytes required to store each data element
<code>mxGetEps (C and Fortran)</code>	Value of eps

<code>mxGetField</code> (C and Fortran)	Field value, given field name and index into structure array
<code>mxGetFieldByNumber</code> (C and Fortran)	Field value, given field number and index into structure array
<code>mxGetFieldNameByNumber</code> (C and Fortran)	Field name, given field number in structure array
<code>mxGetFieldNumber</code> (C and Fortran)	Field number, given field name in structure array
<code>mxGetImagData</code> (C and Fortran)	Pointer to imaginary data of <code>mxArray</code>
<code>mxGetInf</code> (C and Fortran)	Value of infinity
<code>mxGetIr</code> (C and Fortran)	<code>ir</code> array of sparse matrix
<code>mxGetJc</code> (C and Fortran)	<code>jc</code> array of sparse matrix
<code>mxGetLogicals</code> (C)	Pointer to logical array data
<code>mxGetM</code> (C and Fortran)	Number of rows in <code>mxArray</code>
<code>mxGetN</code> (C and Fortran)	Number of columns in <code>mxArray</code>
<code>mxGetNaN</code> (C and Fortran)	Value of NaN (Not-a-Number)
<code>mxGetNumberOfDimensions</code> (C and Fortran)	Number of dimensions in <code>mxArray</code>
<code>mxGetNumberOfElements</code> (C and Fortran)	Number of elements in <code>mxArray</code>
<code>mxGetNumberOfFields</code> (C and Fortran)	Number of fields in structure <code>mxArray</code>
<code>mxGetNzmax</code> (C and Fortran)	Number of elements in <code>ir</code> , <code>pr</code> , and <code>pi</code> arrays
<code>mxGetPi</code> (C and Fortran)	Imaginary data elements in <code>mxArray</code>
<code>mxGetPr</code> (C and Fortran)	Real data elements in <code>mxArray</code>
<code>mxGetScalar</code> (C and Fortran)	Real component of first data element in <code>mxArray</code>
<code>mxGetString</code> (C and Fortran)	Copy string <code>mxArray</code> to C-style string

<code>mxIsCell</code> (C and Fortran)	Determine whether input is cell <code>mxArray</code>
<code>mxIsChar</code> (C and Fortran)	Determine whether input is string <code>mxArray</code>
<code>mxIsClass</code> (C and Fortran)	Determine whether <code>mxArray</code> is member of specified class
<code>mxIsComplex</code> (C and Fortran)	Determine whether data is complex
<code>mxIsDouble</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as double-precision, floating-point numbers
<code>mxIsEmpty</code> (C and Fortran)	Determine whether <code>mxArray</code> is empty
<code>mxIsFinite</code> (C and Fortran)	Determine whether input is finite
<code>mxIsFromGlobalWS</code> (C and Fortran)	Determine whether <code>mxArray</code> was copied from MATLAB global workspace
<code>mxIsInf</code> (C and Fortran)	Determine whether input is infinite
<code>mxIsInt16</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as signed 16-bit integers
<code>mxIsInt32</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as signed 32-bit integers
<code>mxIsInt64</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as signed 64-bit integers
<code>mxIsInt8</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as signed 8-bit integers
<code>mxIsLogical</code> (C and Fortran)	Determine whether <code>mxArray</code> is of class <code>mxLogical</code>
<code>mxIsLogicalScalar</code> (C)	Determine whether scalar <code>mxArray</code> is of class <code>mxLogical</code>

<code>mxIsLogicalScalarTrue</code> (C)	Determine whether scalar <code>mxArray</code> of class <code>mxLogical</code> is true
<code>mxIsNaN</code> (C and Fortran)	Determine whether input is NaN (Not-a-Number)
<code>mxIsNumeric</code> (C and Fortran)	Determine whether <code>mxArray</code> is numeric
<code>mxIsSingle</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as single-precision, floating-point numbers
<code>mxIsSparse</code> (C and Fortran)	Determine whether input is sparse <code>mxArray</code>
<code>mxIsStruct</code> (C and Fortran)	Determine whether input is structure <code>mxArray</code>
<code>mxIsUint16</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as unsigned 16-bit integers
<code>mxIsUint32</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as unsigned 32-bit integers
<code>mxIsUint64</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as unsigned 64-bit integers
<code>mxIsUint8</code> (C and Fortran)	Determine whether <code>mxArray</code> represents data as unsigned 8-bit integers
<code>mxMalloc</code> (C and Fortran)	Allocate dynamic memory using MATLAB memory manager
<code>mxRealloc</code> (C and Fortran)	Reallocate memory
<code>mxRemoveField</code> (C and Fortran)	Remove field from structure array
<code>mxSetCell</code> (C and Fortran)	Set value of one cell of <code>mxArray</code>
<code>mxSetClassName</code> (C)	Convert structure array to MATLAB object array
<code>mxSetData</code> (C and Fortran)	Set pointer to data

<code>mxSetDimensions</code> (C and Fortran)	Modify number of dimensions and size of each dimension
<code>mxSetField</code> (C and Fortran)	Set structure array field, given field name and index
<code>mxSetFieldByNumber</code> (C and Fortran)	Set structure array field, given field number and index
<code>mxSetImagData</code> (C and Fortran)	Set imaginary data pointer for <code>mxArray</code>
<code>mxSetIr</code> (C and Fortran)	Set <code>ir</code> array of sparse <code>mxArray</code>
<code>mxSetJc</code> (C and Fortran)	Set <code>jc</code> array of sparse <code>mxArray</code>
<code>mxSetM</code> (C and Fortran)	Set number of rows in <code>mxArray</code>
<code>mxSetN</code> (C and Fortran)	Set number of columns in <code>mxArray</code>
<code>mxSetNzmax</code> (C and Fortran)	Set storage space for nonzero elements
<code>mxSetPi</code> (C and Fortran)	Set new imaginary data for <code>mxArray</code>
<code>mxSetPr</code> (C and Fortran)	Set new real data for <code>mxArray</code>

## MEX-Files

<code>mexAtExit</code> (C and Fortran)	Register function to call when MEX-function is cleared or MATLAB terminates
<code>mexCallMATLAB</code> (C and Fortran)	Call MATLAB function or user-defined M-file or MEX-file
<code>mexErrMsgIdAndTxt</code> (C and Fortran)	Issue error message with identifier and return to MATLAB prompt
<code>mexErrMsgTxt</code> (C and Fortran)	Issue error message and return to MATLAB prompt
<code>mexEvalString</code> (C and Fortran)	Execute MATLAB command in caller's workspace

<code>mexFunction</code> (C and Fortran)	Entry point to C MEX-file
<code>mexFunctionName</code> (C and Fortran)	Name of current MEX-function
<code>mexGet</code> (C)	Value of specified Handle Graphics® property
<code>mexGetVariable</code> (C and Fortran)	Copy of variable from specified workspace
<code>mexGetVariablePtr</code> (C and Fortran)	Read-only pointer to variable from another workspace
<code>mexIsGlobal</code> (C and Fortran)	Determine whether <code>mxArray</code> has global scope
<code>mexIsLocked</code> (C and Fortran)	Determine whether MEX-file is locked
<code>mexLock</code> (C and Fortran)	Prevent MEX-file from being cleared from memory
<code>mexMakeArrayPersistent</code> (C and Fortran)	Make <code>mxArray</code> persist after MEX-file completes
<code>mexMakeMemoryPersistent</code> (C and Fortran)	Make allocated memory MATLAB persist after MEX-function completes
<code>mexPrintf</code> (C and Fortran)	ANSI C <code>printf</code> -style output routine
<code>mexPutVariable</code> (C and Fortran)	Copy <code>mxArray</code> from MEX-function into specified workspace
<code>mexSet</code> (C)	Set value of specified Handle Graphics property
<code>mexSetTrapFlag</code> (C and Fortran)	Control response of <code>mexCallMATLAB</code> to errors
<code>mexUnlock</code> (C and Fortran)	Allow MEX-file to be cleared from memory
<code>mexWarnMsgIdAndTxt</code> (C and Fortran)	Issue warning message with identifier
<code>mexWarnMsgTxt</code> (C and Fortran)	Issue warning message



## MATLAB Engine

<code>engClose</code> (C and Fortran)	Quit MATLAB engine session
<code>engEvalString</code> (C and Fortran)	Evaluate expression in string
<code>engGetVariable</code> (C and Fortran)	Copy variable from MATLAB engine workspace
<code>engGetVisible</code> (C)	Determine visibility of MATLAB engine session
<code>engOpen</code> (C and Fortran)	Start MATLAB engine session
<code>engOpenSingleUse</code> (C)	Start MATLAB engine session for single, nonshared use
<code>engOutputBuffer</code> (C and Fortran)	Specify buffer for MATLAB output
<code>engPutVariable</code> (C and Fortran)	Put variables into MATLAB engine workspace
<code>engSetVisible</code> (C)	Show or hide MATLAB engine session



# Functions — Alphabetical List

---

# engClose (C and Fortran)

---

**Purpose** Quit MATLAB engine session

**C Syntax**

```
#include "engine.h"
int engClose(Engine *ep);
```

**Fortran Syntax**

```
integer*4 function engClose(ep)
mwPointer ep
```

**Arguments**

ep  
Engine pointer

**Description**

This routine allows you to quit a MATLAB engine session. engClose sends a quit command to the MATLAB engine session and closes the connection. It returns 0 on success, and 1 otherwise. Possible failure includes attempting to terminate a MATLAB engine session that was already terminated.

**C Examples**

**UNIX**

See engdemo.c in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

**Windows**

See engwindemo.c in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

**Fortran Examples**

See fengdemo.F in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

**See Also** engOpen

**Purpose** Evaluate expression in string

**C Syntax**

```
#include "engine.h"
int engEvalString(Engine *ep,const char *string);
```

**Fortran Syntax**

```
integer*4 function engEvalString(ep, string)
mwPointer ep
character*(*) string
```

**Arguments**

ep  
Engine pointer

string  
String to execute

**Description** engEvalString evaluates the expression contained in string for the MATLAB engine session, ep, previously started by engOpen. It returns a nonzero value if the MATLAB session is no longer running, and zero otherwise.

On UNIX systems, engEvalString sends commands to MATLAB by writing down a pipe connected to the MATLAB *stdin*. Any output resulting from the command that ordinarily appears on the screen is read back from *stdout* into the buffer defined by engOutputBuffer.

To turn off output buffering in C, use

```
engOutputBuffer(ep, NULL, 0);
```

To turn off output buffering in Fortran, use

```
engOutputBuffer(ep, '')
```

Under Windows on a PC, engEvalString communicates with MATLAB using a Component Object Model (COM) interface.

## engEvalString (C and Fortran)

---

### **C Examples**

#### **UNIX**

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

#### **Windows**

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

### **Fortran Examples**

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

### **See Also**

`engOpen`, `engOutputBuffer`

# engGetVariable (C and Fortran)

---

<b>Purpose</b>	Copy variable from MATLAB engine workspace
<b>C Syntax</b>	<pre>#include "engine.h" mxArray *engGetVariable(Engine *ep, const char *name);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function engGetVariable(ep, name) mwPointer ep character*(*) name</pre>
<b>Arguments</b>	<p>ep Engine pointer</p> <p>name Name of mxArray to get from MATLAB</p>
<b>Description</b>	<p>engGetVariable reads the named mxArray from the MATLAB engine session associated with ep and returns a pointer to a newly allocated mxArray structure, or NULL if the attempt fails. engGetVariable fails if the named variable does not exist.</p> <p>Be careful in your code to free the mxArray created by this routine when you are finished with it.</p>
<b>C Examples</b>	<p><b>UNIX</b></p> <p>See engdemo.c in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.</p> <p><b>Windows</b></p> <p>See engwindemo.c in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.</p>
<b>See Also</b>	engPutVariable

# engGetVisible (C)

---

**Purpose** Determine visibility of MATLAB engine session

**C Syntax**

```
#include "engine.h"
int engGetVisible(Engine *ep, bool *value);
```

**Arguments**

ep  
Engine pointer

value  
Pointer to value returned from engGetVisible

**Description**

**Windows Only**

engGetVisible returns the current visibility setting for MATLAB engine session, ep. A *visible* engine session runs in a window on the Windows desktop, thus making the engine available for user interaction. An invisible session is hidden from the user by removing it from the desktop.

engGetVisible returns 0 on success, and 1 otherwise.

**Examples** The following code opens engine session ep and disables its visibility.

```
Engine *ep;
bool vis;

ep = engOpen(NULL);
engSetVisible(ep, 0);
```

To determine the current visibility setting, use

```
engGetVisible(ep, &vis);
```

**See Also** engSetVisible



<b>Purpose</b>	Start MATLAB engine session
<b>C Syntax</b>	<pre>#include "engine.h" Engine *engOpen(const char *startcmd);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function engOpen(startcmd) character*(*) startcmd</pre>
<b>Arguments</b>	<p>startcmd</p> <p>String to start the MATLAB process. On Windows, the startcmd string must be NULL.</p>
<b>Returns</b>	A pointer to an engine handle.
<b>Description</b>	<p>This routine allows you to start a MATLAB process for the purpose of using MATLAB as a computational engine.</p> <p>engOpen(startcmd) starts a MATLAB process using the command specified in the string startcmd, establishes a connection, and returns a unique engine identifier, or NULL if the open fails.</p> <p>On UNIX systems, if startcmd is NULL or the empty string, engOpen starts MATLAB on the current host using the command matlab. If startcmd is a hostname, engOpen starts MATLAB on the designated host by embedding the specified hostname string into the larger string:</p> <pre>"rsh hostname \"/bin/csh -c 'setenv DISPLAY\ hostname:0; matlab'\\""</pre> <p>If startcmd is any other string (has white space in it, or nonalphanumeric characters), the string is executed literally to start MATLAB.</p> <p>On UNIX systems, engOpen performs the following steps:</p> <ol style="list-style-type: none"><li>1 Creates two pipes.</li></ol>

## engOpen (C and Fortran)

---

- 2 Forks a new process and sets up the pipes to pass *stdin* and *stdout* from MATLAB (parent) to two file descriptors in the engine program (child).
- 3 Executes a command to run MATLAB (*rsh* for remote execution).

Under Windows on a PC, `engOpen` opens a COM channel to MATLAB. This starts the MATLAB that was registered during installation. If you did not register during installation, on the command line you can enter the command

```
matlab /regserver
```

See “Introducing MATLAB COM Integration” for additional details.

### C Examples

#### UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

#### Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

### Fortran Examples

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

**Purpose** Start MATLAB engine session for single, nonshared use

**C Syntax**

```
#include "engine.h"
Engine *engOpenSingleUse(const char *startcmd, void *dcom,
    int *retstatus);
```

**Arguments**

`startcmd`  
String to start MATLAB process. On Windows, the `startcmd` string must be NULL.

`dcom`  
Reserved for future use; must be NULL.

`retstatus`  
Return status; possible cause of failure.

## **Description** **Windows**

This routine allows you to start multiple MATLAB processes for the purpose of using MATLAB as a computational engine. `engOpenSingleUse` starts a MATLAB process, establishes a connection, and returns a unique engine identifier, or NULL if the open fails. `engOpenSingleUse` starts a new MATLAB process each time it is called.

`engOpenSingleUse` opens a COM channel to MATLAB. This starts the MATLAB that was registered during installation. If you did not register during installation, on the command line you can enter the command

```
matlab /regserver
```

`engOpenSingleUse` allows single-use instances of a MATLAB engine server. `engOpenSingleUse` differs from `engOpen`, which allows multiple users to use the same MATLAB engine server.

See “Introducing MATLAB COM Integration” for additional details.

## **UNIX**

This routine is not supported and simply returns.

# engOutputBuffer (C and Fortran)

---

**Purpose** Specify buffer for MATLAB output

**C Syntax**

```
#include "engine.h"
int engOutputBuffer(Engine *ep, char *p, int n);
```

**Fortran Syntax**

```
integer*4 function engOutputBuffer(ep, p)
mwPointer ep
character*n p
```

**Arguments**

ep	Engine pointer
p	Pointer to character buffer
n	Length of buffer p

**Description** engOutputBuffer defines a character buffer for engEvalString to return any output that ordinarily appears on the screen. It returns 1 if you pass it a NULL engine pointer. Otherwise, it returns 0.

The default behavior of engEvalString is to discard any standard output caused by the command it is executing. A call to engOutputBuffer with a buffer of nonzero length tells any subsequent calls to engEvalString to save output in the character buffer pointed to by p.

To turn off output buffering in C, use

```
engOutputBuffer(ep, NULL, 0);
```

To turn off output buffering in Fortran, use

```
engOutputBuffer(ep, '')
```

---

**Note** The buffer returned by `engEvalString` is not guaranteed to be NULL terminated.

---

## **C Examples**

### **UNIX**

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

### **Windows**

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

## **Fortran Examples**

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

## **See Also**

`engOpen`, `engEvalString`

# engPutVariable (C and Fortran)

---

**Purpose** Put variables into MATLAB engine workspace

**C Syntax**

```
#include "engine.h"
int engPutVariable(Engine *ep, const char *name, const mxArray
    *pm);
```

**Fortran Syntax**

```
integer*4 function engPutVariable(ep, name, pm)
mwPointer ep, pm
character*(*) name
```

**Arguments**

ep  
Engine pointer

name  
Name given to the mxArray in the engine's workspace

pm  
mxArray pointer

**Description**

engPutVariable writes mxArray pm to the engine ep, giving it the variable name name. If the mxArray does not exist in the workspace, it is created. If an mxArray with the same name already exists in the workspace, the existing mxArray is replaced with the new mxArray.

engPutVariable returns 0 if successful and 1 if an error occurs.

**C Examples**

**UNIX**

See engdemo.c in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

**Windows**

See engwindemo.c in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

### **See Also**

`engGetVariable`

# engSetVisible (C)

---

<b>Purpose</b>	Show or hide MATLAB engine session
<b>C Syntax</b>	<pre>#include "engine.h" int engSetVisible(Engine *ep, bool value);</pre>
<b>Arguments</b>	<p>ep Engine pointer</p> <p>value Value to set the Visible property to. Set value to 1 to make the engine window visible, or to 0 to make it invisible.</p>
<b>Description</b>	<p><b>Windows Only</b></p> <p>engSetVisible makes the window for the MATLAB engine session, ep, either visible or invisible on the Windows desktop. You can use this function to enable or disable user interaction with the MATLAB engine session.</p> <p>engSetVisible returns 0 on success, and 1 otherwise.</p>
<b>Examples</b>	<p>The following code opens engine session ep and disables its visibility.</p> <pre>Engine *ep; bool vis;  ep = engOpen(NULL); engSetVisible(ep, 0);</pre> <p>To determine the current visibility setting, use</p> <pre>engGetVisible(ep, &amp;vis);</pre>
<b>See Also</b>	engGetVisible



<b>Purpose</b>	Close MAT-file
<b>C Syntax</b>	<pre>#include "mat.h" int matClose(MATFile *mfp);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function matClose(mfp) mwPointer mfp</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p>
<b>Returns</b>	EOF in C (-1 in Fortran) for a write error, and 0 if successful.
<b>Description</b>	matClose closes the MAT-file associated with mfp.
<b>C Examples</b>	See matcreat.c and matdgns.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.
<b>Fortran Examples</b>	See matdemo1.F and matdemo2.F in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use this MAT-file routine in a Fortran program.

# matDeleteVariable (C and Fortran)

---

<b>Purpose</b>	Delete named mxArray from MAT-file
<b>C Syntax</b>	<pre>#include "mat.h" int matDeleteVariable(MATFile *mfp, const char *name);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function matDeleteVariable(mfp, name) mwPointer mfp character*(*) name</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p> <p>name     Name of mxArray to delete</p>
<b>Returns</b>	0 if successful, and nonzero otherwise.
<b>Description</b>	matDeleteVariable deletes the named mxArray from the MAT-file pointed to by mfp.
<b>C Examples</b>	See matcreat.c and matdgns.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

<b>Purpose</b>	Directory of mxArray's in MAT-file
<b>C Syntax</b>	<pre>#include "mat.h" char **matGetDir(MATFile *mfp, int *num);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function matGetDir(mfp, num) mwPointer mfp integer*4 num</pre>
<b>Arguments</b>	<p>mfp Pointer to MAT-file information</p> <p>num Address of the variable to contain the number of mxArray's in the MAT-file</p>
<b>Returns</b>	<p>A pointer to an internal array containing pointers to the names of the mxArray's in the MAT-file pointed to by mfp. In C, each name is a NULL-terminated string. The length of the internal array (number of mxArray's in the MAT-file) is placed into num. If num is zero, mfp contains no arrays.</p> <p>matGetDir returns NULL in C (0 in Fortran) and sets num to a negative number if it fails.</p>
<b>Description</b>	<p>This routine allows you to get a list of the names of the mxArray's contained within a MAT-file.</p> <p>The internal array of strings that matGetDir returns is allocated using a single mxMalloc and must be freed using mxFree when you are finished with it.</p> <p>MATLAB variable names can be up to length mxMAXNAM, where mxMAXNAM is defined in the C header file matrix.h.</p>
<b>C Examples</b>	See matcreat.c and matdgns.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

## matGetDir (C and Fortran)

---

### **Fortran Examples**

See `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this MAT-file routine in a Fortran program.

<b>Purpose</b>	File pointer to MAT-file
<b>C Syntax</b>	<pre>#include "mat.h" FILE *matGetFp(MATFile *mfp);</pre>
<b>Arguments</b>	mfp Pointer to MAT-file information
<b>Returns</b>	A C file handle to the MAT-file with handle mfp. Returns NULL if mfp is a handle to a MAT-file in HDF5-based format.
<b>Description</b>	Use matGetFp to obtain a C file handle to a MAT-file. This can be useful for using standard C library routines like ferror() and feof() to investigate error situations.
<b>Examples</b>	See matcreat.c and matdgn.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

# matGetNextVariable (C and Fortran)

---

<b>Purpose</b>	Read next mxArray from MAT-file
<b>C Syntax</b>	<pre>#include "mat.h" mxArray *matGetNextVariable(MATFile *mfp, const char *name);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function matGetNextVariable(mfp, name) mwPointer mfp character*(*) name</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p> <p>name     Address of the variable to contain the mxArray name</p>
<b>Returns</b>	<p>A pointer to a newly allocated mxArray structure representing the next mxArray from the MAT-file pointed to by mfp. The function returns the name of the mxArray in name.</p> <p>matGetNextVariable returns NULL in C (0 in Fortran) when the end-of-file is reached or if there is an error condition. In C, you can use feof and ferror from the Standard C Library to determine status.</p>
<b>Description</b>	<p>matGetNextVariable allows you to step sequentially through a MAT-file and read all the mxArrays in a single pass. The function reads and returns the next mxArray from the MAT-file pointed to by mfp.</p> <p>Use matGetNextVariable immediately after opening the MAT-file with matOpen and not in conjunction with other MAT-file routines. Otherwise, the concept of the <i>next</i> mxArray is undefined.</p> <p>Free the memory used by the mxArray created by this routine when you are finished with it.</p>
<b>C Examples</b>	See matcreat.c and matdgns.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

# matGetNextVariableInfo (C and Fortran)

---

<b>Purpose</b>	Load array header information only
<b>C Syntax</b>	<pre>#include "mat.h" mxArray *matGetNextVariableInfo(MATFile *mfp, const char *name);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function matGetNextVariableInfo(mfp, name) mwPointer mfp character*(*) name</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p> <p>name     Address of the variable to contain the mxArray name</p>
<b>Returns</b>	<p>A pointer to a newly allocated mxArray structure representing header information for the next mxArray from the MAT-file pointed to by mfp. The function returns the name of the mxArray in name.</p> <p>matGetNextVariableInfo returns NULL in C (0 in Fortran) when the end-of-file is reached or if there is an error condition. In C, you can use feof and ferror from the Standard C Library to determine status.</p>
<b>Description</b>	<p>matGetNextVariableInfo loads only the array header information, including everything except pr, pi, ir, and jc, from the file's current file offset.</p> <p>If pr, pi, ir, and jc are set to nonzero values when loaded with matGetVariable, matGetNextVariableInfo sets them to -1 instead. These headers are for informational use only and should <i>never</i> be passed back to MATLAB or saved to MAT-files.</p> <p>Free the memory used by the mxArray created by this routine when you are finished with it.</p>
<b>C Examples</b>	See matcreat.c and matdgn.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

## matGetNextVariableInfo (C and Fortran)

---

### **See Also**

`matGetNextVariable`, `matGetVariableInfo`



# matGetVariable (C and Fortran)

---

<b>Purpose</b>	Read mxArray from MAT-files
<b>C Syntax</b>	<pre>#include "mat.h" mxArray *matGetVariable(MATFile *mfp, const char *name);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function matGetVariable(mfp, name) mwPointer mfp character*(*) name</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p> <p>name     Name of mxArray to get from MAT-file</p>
<b>Returns</b>	<p>A pointer to a newly allocated mxArray structure representing the mxArray named by name from the MAT-file pointed to by mfp.</p> <p>matGetVariable returns NULL in C (0 in Fortran) if the attempt to return the mxArray named by name fails.</p>
<b>Description</b>	<p>This routine allows you to copy an mxArray out of a MAT-file.</p> <p>Free the memory used by the mxArray created by this routine when you are finished with it.</p>
<b>C Examples</b>	See matcreat.c and matdgn.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

# matGetVariableInfo (C and Fortran)

---

<b>Purpose</b>	Load array header information only
<b>C Syntax</b>	<pre>#include "mat.h" mxArray *matGetVariableInfo(MATFile *mfp, const char *name);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function matGetVariableInfo(mfp, name); mwPointer mfp character*(*) name</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p> <p>name     Name of mxArray to get from MAT-file</p>
<b>Returns</b>	<p>A pointer to a newly allocated mxArray structure representing header information for the mxArray named by name from the MAT-file pointed to by mfp.</p> <p>matGetVariableInfo returns NULL in C (0 in Fortran) if the attempt to return header information for the mxArray named by name fails.</p>
<b>Description</b>	<p>matGetVariableInfo loads only the array header information, including everything except pr, pi, ir, and jc. It recursively creates the cells and structures through their leaf elements, but does not include pr, pi, ir, and jc.</p> <p>If pr, pi, ir, and jc are set to nonzero values when loaded with matGetVariable, matGetVariableInfo sets them to -1 instead. These headers are for informational use only and should <i>never</i> be passed back to MATLAB or saved to MAT-files.</p> <p>Free the memory used by the mxArray created by this routine when you are finished with it.</p>
<b>C Examples</b>	See matcreat.c and matdgn.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

## matGetVariableInfo (C and Fortran)

---

**See Also**      `matGetVariable`

# matOpen (C and Fortran)

---

## Purpose

Open MAT-file

## C Syntax

```
#include "mat.h"  
MATFile *matOpen(const char *filename, const char *mode);
```

## Fortran Syntax

```
mwPointer function matOpen(filename, mode)  
character*(*) filename, mode
```

## Arguments

filename

Name of file to open

mode

File opening mode. Valid values for mode are listed in the following table.

r	Opens file for reading only; determines the current version of the MAT-file by inspecting the files and preserves the current version.
u	Opens file for update, both reading and writing, but does not create the file if the file does not exist (equivalent to the r+ mode of fopen); determines the current version of the MAT-file by inspecting the files and preserves the current version.
w	Opens file for writing only; deletes previous contents, if any.
w4	Creates a Level 4 MAT-file, compatible with MATLAB Versions 4 and earlier.
wL	Opens file for writing character data using the default character set for your system. The resulting MAT-file can be read with MATLAB Version 6 or 6.5.  If you do not use the wL mode switch, MATLAB writes character data to the MAT-file using Unicode character encoding by default.

wz	Opens file for writing compressed data.
w7.3	Creates a MAT-file in an HDF5-based format that can store objects occupy more than 2 GB.

**Returns** A file handle, or NULL in C (0 in Fortran) if the open fails.

**Description** This routine opens a MAT-file for reading and writing.  
See “Writing Character Data” in the External Interfaces documentation for more information on how MATLAB uses character encodings.

**C Examples** See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

**Fortran Examples** See `matdemo1.F` and `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a Fortran program.

# matPutVariable (C and Fortran)

---

**Purpose** Write mxArray to MAT-files

**C Syntax**

```
#include "mat.h"
int matPutVariable(MATFile *mfp, const char *name, const mxArray
    *pm);
```

**Fortran Syntax**

```
integer*4 function matPutVariable(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

**Arguments**

mfp  
Pointer to MAT-file information

name  
Name of mxArray to put into MAT-file

pm  
mxArray pointer

**Returns** 0 if successful and nonzero if an error occurs. In C, use feof and ferror from the Standard C Library along with matGetFp to determine status.

**Description** This routine allows you to put an mxArray into a MAT-file. matPutVariable writes mxArray pm to the MAT-file mfp. If the mxArray does not exist in the MAT-file, it is appended to the end. If an mxArray with the same name already exists in the file, the existing mxArray is replaced with the new mxArray by rewriting the file. The size of the new mxArray can be different from the existing mxArray.

**C Examples** See matcreat.c and matdgn.c in the eng\_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

# matPutVariableAsGlobal (C and Fortran)

---

<b>Purpose</b>	Put mxArray into MAT-files as originating from global workspace
<b>C Syntax</b>	<pre>#include "mat.h" int matPutVariableAsGlobal(MATFile *mfp, const char *name, const     mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function matPutVariableAsGlobal(mfp, name, pm) mwPointer mfp, pm character*(*) name</pre>
<b>Arguments</b>	<p>mfp     Pointer to MAT-file information</p> <p>name     Name of mxArray to put into MAT-file</p> <p>pm     mxArray pointer</p>
<b>Returns</b>	0 if successful and nonzero if an error occurs. In C, use feof and ferrord from the Standard C Library with matGetFp to determine status.
<b>Description</b>	<p>This routine puts an mxArray into a MAT-file. matPutVariableAsGlobal is similar to matPutVariable, except that the array, when loaded by MATLAB, is placed into the global workspace and a reference to it is set in the local workspace. If you write to a MATLAB 4 format file, matPutVariableAsGlobal does not load it as global and has the same effect as matPutVariable.</p> <p>matPutVariableAsGlobal writes mxArray pm to the MAT-file mfp. If the mxArray does not exist in the MAT-file, it is appended to the end. If an mxArray with the same name already exists in the file, the existing mxArray is replaced with the new mxArray by rewriting the file. The size of the new mxArray can be different from the existing mxArray.</p>

## matPutVariableAsGlobal (C and Fortran)

---

### **C Examples**

See `matcreat.c` and `matdgn.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.



<b>Purpose</b>	Register function to call when MEX-function is cleared or MATLAB terminates
<b>C Syntax</b>	<pre>#include "mex.h" int mexAtExit(void (*ExitFcn)(void));</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mexAtExit(ExitFcn) subroutine ExitFcn()</pre>
<b>Arguments</b>	ExitFcn Pointer to function you want to run on exit
<b>Returns</b>	Always returns 0.
<b>Description</b>	<p>Use mexAtExit to register a function to be called just before the MEX-function is cleared or MATLAB is terminated. mexAtExit gives your MEX-function a chance to perform tasks such as freeing persistent memory and closing files. Typically, the named ExitFcn performs tasks like closing streams or sockets.</p> <p>Each MEX-function can register only one active exit function at a time. If you call mexAtExit more than once, MATLAB uses the ExitFcn from the more recent mexAtExit call as the exit function.</p> <p>If a MEX-function is locked, all attempts to clear the MEX-file will fail. Consequently, if a user attempts to clear a locked MEX-file, MATLAB does not call the ExitFcn.</p> <p>In Fortran, you must declare the ExitFcn as external in the Fortran routine that calls mexAtExit if it is not within the scope of the file.</p>
<b>C Examples</b>	See mexatexit.c in the mex subdirectory of the examples directory.
<b>See Also</b>	mexLock, mexUnlock, mexSetTrapFlag

# mexCallMATLAB (C and Fortran)

---

## Purpose

Call MATLAB function or user-defined M-file or MEX-file

## C Syntax

```
#include "mex.h"
int mexCallMATLAB(int nlhs, mxArray *plhs[], int nrhs,
    mxArray *prhs[], const char *name);
```

## Fortran Syntax

```
integer*4 function mexCallMATLAB(nlhs, plhs, nrhs, prhs, name)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) name
```

## Arguments

nlhs

Number of desired output arguments. This value must be less than or equal to 50.

plhs

Array of pointers to `mxArrays`. The called command puts pointers to the resultant `mxArrays` into `plhs` and allocates dynamic memory to store the resultant `mxArrays`. By default, MATLAB automatically deallocates this dynamic memory when you clear the MEX-file. However, if heap space is at a premium, you may want to call `mxDestroyArray` as soon as you are finished with the `mxArrays` that `plhs` points to.

nrhs

Number of input arguments. This value must be less than or equal to 50.

prhs

Array of pointers to input arguments.

name

Character string containing the name of the MATLAB built-in, operator, M-file, or MEX-file that you are calling. If `name` is an operator, just place the operator inside a pair of single quotes, for example, '+'.

## Returns

0 if successful, and a nonzero value if unsuccessful.

## Description

Call `mexCallMATLAB` to invoke internal MATLAB numeric functions, MATLAB operators, M-files, or other MEX-files. See `mexFunction` for a complete description of the arguments.

By default, if name detects an error, MATLAB terminates the MEX-file and returns control to the MATLAB prompt. If you want a different error behavior, turn on the trap flag by calling `mexSetTrapFlag`.

It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, if you create an M-file that returns two variables but assigns only one of them a value,

```
function [a,b]=foo(c)
a=2*c;
```

you get this warning message in MATLAB:

```
Warning: One or more output arguments not assigned
during call to 'foo'.
```

MATLAB assigns output `b` to an empty matrix. If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is given type `mxUNKNOWN_CLASS`.

## C Examples

See `mexcallmatlab.c` in the `mex` subdirectory of the `examples` directory.

Additional examples:

- `sincall.c` in the `refbook` subdirectory of the `examples` directory
- `mexevalstring.c` and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory
- `mxcreatecellmatrix.c` and `mxisclass.c` in the `mx` subdirectory of the `examples` directory

## See Also

`mexFunction`, `mexSetTrapFlag`

# mexErrMsgIdAndTxt (C and Fortran)

---

**Purpose** Issue error message with identifier and return to MATLAB prompt

**C Syntax**

```
#include "mex.h"
void mexErrMsgIdAndTxt(const char *errorid,
const char *errmsg, ...);
```

**Fortran Syntax**

```
subroutine mexErrMsgIdAndTxt(errorid, errmsg)
character*(*) errorid, errmsg
```

**Arguments**

**errorid**  
String containing a MATLAB message identifier. See “Message Identifiers” in the MATLAB documentation for information on this topic.

**errmsg**  
String containing the error message to be displayed. In C, the string may include formatting conversion characters, such as those used with the ANSI C `sprintf` function.

...

In C, any additional arguments needed to translate formatting conversion characters used in `errmsg`. Each conversion character in `errmsg` is converted to one of these values.

**Description** Call `mexErrMsgIdAndTxt` to write an error message and its corresponding identifier to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgIdAndTxt` does not clear the MEX-file from memory. Consequently, `mexErrMsgIdAndTxt` does not invoke the function registered through `mexAtExit`.

If your application called `mxCalloc` or one of the `mxCreate` routines to allocate memory, `mexErrMsgIdAndTxt` automatically frees the allocated memory.

---

**Note** If you get warnings when using `mexErrMsgIdAndTxt`, you may have a memory management compatibility problem. For more information, see “Memory Management Compatibility Issues” in the External Interfaces documentation.

---

### See Also

`mexErrMsgTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

# mexErrMsgTxt (C and Fortran)

---

**Purpose** Issue error message and return to MATLAB prompt

**C Syntax**

```
#include "mex.h"
void mexErrMsgTxt(const char *errmsg);
```

**Fortran Syntax**

```
subroutine mexErrMsgTxt(errormsg)
character*(*) errmsg
```

**Arguments** `errmsg`  
String containing the error message to be displayed

**Description** Call `mexErrMsgTxt` to write an error message to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgTxt` does not clear the MEX-file from memory. Consequently, `mexErrMsgTxt` does not invoke the function registered through `mexAtExit`.

If your application called `mxCalloc` or one of the `mxCreat` routines to allocate memory, `mexErrMsgTxt` automatically frees the allocated memory.

---

**Note** If you get warnings when using `mexErrMsgTxt`, you may have a memory management compatibility problem. For more information, see “Memory Management Compatibility Issues”.

---

**C Examples** See `xtimesy.c` in the `refbook` subdirectory of the `examples` directory. For additional examples, see `convec.c`, `findnz.c`, `fulltosparse.c`, `phonebook.c`, `revord.c`, and `timestwo.c` in the `refbook` subdirectory of the `examples` directory.

**See Also** `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

<b>Purpose</b>	Execute MATLAB command in caller's workspace
<b>C Syntax</b>	<pre>#include "mex.h" int mexEvalString(const char *command);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mexEvalString(command) character*(*) command</pre>
<b>Arguments</b>	command A string containing the MATLAB command to execute
<b>Returns</b>	0 if successful, and a nonzero value if unsuccessful.
<b>Description</b>	<p>Call <code>mexEvalString</code> to invoke a MATLAB command in the workspace of the caller.</p> <p><code>mexEvalString</code> and <code>mexCallMATLAB</code> both execute MATLAB commands. However, <code>mexCallMATLAB</code> provides a mechanism for returning results (left-hand side arguments) back to the MEX-file; <code>mexEvalString</code> provides no way for return values to be passed back to the MEX-file.</p> <p>All arguments that appear to the right of an equal sign in the command string must already be current variables of the caller's workspace.</p>
<b>Examples</b>	See <code>mexevalstring.c</code> in the <code>mex</code> subdirectory of the <code>examples</code> directory.
<b>See Also</b>	<code>mexCallMATLAB</code>

# mexFunction (C and Fortran)

---

<b>Purpose</b>	Entry point to C MEX-file
<b>C Syntax</b>	<pre>#include "mex.h" void mexFunction(int nlhs, mxArray *plhs[], int nrhs,                  const mxArray *prhs[]);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mexFunction(nlhs, plhs, nrhs, prhs) integer*4 nlhs, nrhs mwPointer plhs(*), prhs(*)</pre>
<b>Arguments</b>	<p><b>nlhs</b> The number of expected output mxArray</p> <p><b>plhs</b> Array of pointers to the expected output mxArray</p> <p><b>nrhs</b> The number of input mxArray</p> <p><b>prhs</b> Array of pointers to the input mxArray. These mxArray are read only and should not be modified by your MEX-file. Changing the data in these mxArray may produce undesired side effects.</p>
<b>Description</b>	<p>mexFunction is not a routine you call. Rather, mexFunction is the name of a function in C (subroutine in Fortran) that you must write in every MEX-file. When you invoke a MEX-function, MATLAB finds and loads the corresponding MEX-file of the same name. MATLAB then searches for a symbol named mexFunction within the MEX-file. If it finds one, it calls the MEX-function using the address of the mexFunction symbol. If MATLAB cannot find a routine named mexFunction inside the MEX-file, it issues an error message.</p> <p>When you invoke a MEX-file, MATLAB automatically seeds nlhs, plhs, nrhs, and prhs with the caller's information. In the syntax of the MATLAB language, functions have the general form</p> $[a,b,c,\dots] = \text{fun}(d,e,f,\dots)$



where the ... denotes more items of the same format. The `a, b, c...` are left-hand side arguments, and the `d, e, f...` are right-hand side arguments. The arguments `nlhs` and `nrhs` contain the number of left-hand side and right-hand side arguments, respectively, with which the MEX-function is called. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is an array whose length is `nlhs`, where your function must set pointers for the returned left-hand side `mxArrays`.

### **C Examples**

See `mexfunction.c` in the `mex` subdirectory of the `examples` directory.

## mexFunctionName (C and Fortran)

---

<b>Purpose</b>	Name of current MEX-function
<b>C Syntax</b>	<pre>#include "mex.h" const char *mexFunctionName(void);</pre>
<b>Fortran Syntax</b>	<pre>character*(*) function mexFunctionName()</pre>
<b>Returns</b>	The name of the current MEX-function.
<b>Description</b>	mexFunctionName returns the name of the current MEX-function.
<b>C Examples</b>	See mexgetarray.c in the mex subdirectory of the examples directory.

<b>Purpose</b>	Value of specified Handle Graphics® property
<b>C Syntax</b>	<pre>#include "mex.h" const mxArray *mexGet(double handle, const char *property);</pre>
<b>Arguments</b>	<p>handle Handle to a particular graphics object</p> <p>property A Handle Graphics property</p>
<b>Returns</b>	The value of the specified property in the specified graphics object on success. Returns NULL on failure. The return argument from mexGet is declared as constant, meaning that it is read only and should not be modified. Changing the data in these mxArray's may produce undesired side effects.
<b>Description</b>	Call mexGet to get the value of the property of a certain graphics object. mexGet is the API equivalent of the MATLAB get function. To set a graphics property value, call mexSet.
<b>Examples</b>	See mexget.c in the mex subdirectory of the examples directory.
<b>See Also</b>	mexSet

# mexGetVariable (C and Fortran)

---

<b>Purpose</b>	Copy of variable from specified workspace						
<b>C Syntax</b>	<pre>#include "mex.h" mxArray *mexGetVariable(const char *workspace, const char     *varname);</pre>						
<b>Fortran Syntax</b>	<pre>mwPointer function mexGetVariable(workspace, varname) character*(*) workspace, varname</pre>						
<b>Arguments</b>	<p>workspace Specifies where <code>mexGetVariable</code> should search in order to find array <code>varname</code>. The possible values are</p> <table><tr><td>base</td><td>Search for the variable in the base workspace.</td></tr><tr><td>caller</td><td>Search for the variable in the caller's workspace.</td></tr><tr><td>global</td><td>Search for the variable in the global workspace.</td></tr></table> <p>varname Name of the variable to copy</p>	base	Search for the variable in the base workspace.	caller	Search for the variable in the caller's workspace.	global	Search for the variable in the global workspace.
base	Search for the variable in the base workspace.						
caller	Search for the variable in the caller's workspace.						
global	Search for the variable in the global workspace.						
<b>Returns</b>	A copy of the variable on success. Returns NULL in C (0 on Fortran) on failure. A common cause of failure is specifying a variable that is not currently in the workspace. Perhaps the variable was in the workspace at one time but has since been cleared.						
<b>Description</b>	Call <code>mexGetVariable</code> to get a copy of the specified variable. The returned <code>mxArray</code> contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned <code>mxArray</code> do not affect the variable in the workspace unless you write the copy back to the workspace with <code>mexPutVariable</code> .						
<b>C Examples</b>	See <code>mexgetarray.c</code> in the <code>mex</code> subdirectory of the <code>examples</code> directory.						
<b>See Also</b>	<code>mexGetVariablePtr</code> , <code>mexPutVariable</code>						

# mexGetVariablePtr (C and Fortran)

---

<b>Purpose</b>	Read-only pointer to variable from another workspace						
<b>C Syntax</b>	<pre>#include "mex.h" const mxArray *mexGetVariablePtr(const char *workspace,     const char *varname);</pre>						
<b>Fortran Syntax</b>	<pre>mwPointer function mexGetVariablePtr(workspace, varname) character*(*) workspace, varname</pre>						
<b>Arguments</b>	<p>workspace Specifies which workspace you want mexGetVariablePtr to search. The possible values are</p> <table><tr><td>base</td><td>Search for the variable in the base workspace.</td></tr><tr><td>caller</td><td>Search for the variable in the caller's workspace.</td></tr><tr><td>global</td><td>Search for the variable in the global workspace.</td></tr></table> <p>varname Name of a variable in another workspace. This is a variable name, not an mxArray pointer.</p>	base	Search for the variable in the base workspace.	caller	Search for the variable in the caller's workspace.	global	Search for the variable in the global workspace.
base	Search for the variable in the base workspace.						
caller	Search for the variable in the caller's workspace.						
global	Search for the variable in the global workspace.						
<b>Returns</b>	A read-only pointer to the mxArray on success. Returns NULL in C (0 in Fortran) on failure.						
<b>Description</b>	<p>Call mexGetVariablePtr to get a read-only pointer to the specified variable, varname, into your MEX-file's workspace. This command is useful for examining an mxArray's data and characteristics. If you need to change data or characteristics, use mexGetVariable (along with mexPutVariable) instead of mexGetVariablePtr.</p> <p>If you simply need to examine data or characteristics, mexGetVariablePtr offers superior performance because the caller needs to pass only a pointer to the array.</p>						

## mexGetVariablePtr (C and Fortran)

---

### **C Examples**

See `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

### **See Also**

`mexGetVariable`

<b>Purpose</b>	Determine whether mxArray has global scope
<b>C Syntax</b>	<pre>#include "matrix.h" bool mexIsGlobal(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mexIsGlobal(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the mxArray has global scope, and logical 0 (false) otherwise.
<b>Description</b>	Use mexIsGlobal to determine whether the specified mxArray has global scope.
<b>C Examples</b>	See mxislogical.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mexGetVariable, mexGetVariablePtr, mexPutVariable, global

# mexIsLocked (C and Fortran)

---

<b>Purpose</b>	Determine whether MEX-file is locked
<b>C Syntax</b>	<pre>#include "mex.h" bool mexIsLocked(void);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mexIsLocked()</pre>
<b>Returns</b>	Logical 1 (true) if the MEX-file is locked; logical 0 (false) if the file is unlocked.
<b>Description</b>	<p>Call <code>mexIsLocked</code> to determine whether the MEX-file is locked. By default, MEX-files are unlocked, meaning that users can clear the MEX-file at any time.</p> <p>To unlock a MEX-file, call <code>mexUnlock</code>.</p>
<b>C Examples</b>	See <code>mexlock.c</code> in the <code>mex</code> subdirectory of the <code>examples</code> directory.
<b>See Also</b>	<code>mexLock</code> , <code>mexMakeArrayPersistent</code> , <code>mexMakeMemoryPersistent</code> , <code>mexUnlock</code>



<b>Purpose</b>	Prevent MEX-file from being cleared from memory
<b>C Syntax</b>	<pre>#include "mex.h" void mexLock(void);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mexLock()</pre>
<b>Description</b>	<p>By default, MEX-files are unlocked, meaning that a user can clear them at any time. Call <code>mexLock</code> to prohibit a MEX-file from being cleared.</p> <p>To unlock a MEX-file, call <code>mexUnlock</code>.</p> <p><code>mexLock</code> increments a lock count. If you call <code>mexLock</code> <code>n</code> times, you must call <code>mexUnlock</code> <code>n</code> times to unlock your MEX-file.</p>
<b>C Examples</b>	See <code>mexlock.c</code> in the <code>mex</code> subdirectory of the <code>examples</code> directory.
<b>See Also</b>	<code>mexIsLocked</code> , <code>mexMakeArrayPersistent</code> , <code>mexMakeMemoryPersistent</code> , <code>mexUnlock</code>

# mexMakeArrayPersistent (C and Fortran)

---

**Purpose** Make mxArray persist after MEX-file completes

**C Syntax**

```
#include "mex.h"
void mexMakeArrayPersistent(mxArray *pm);
```

**Fortran Syntax**

```
subroutine mexMakeArrayPersistent(pm)
mwPointer pm
```

**Arguments** pm  
Pointer to an mxArray created by an mxCreate\* function

**Description** By default, mxArrays allocated by mxCreate\* functions are not persistent. The MATLAB memory management facility automatically frees nonpersistent mxArrays when the MEX-function finishes. If you want the mxArray to persist through multiple invocations of the MEX-function, you must call mexMakeArrayPersistent.

---

**Note** If you create a persistent mxArray, you are responsible for destroying it when the MEX-file is cleared. If you do not destroy a persistent mxArray, MATLAB leaks memory. See mexAtExit to see how to register a function that gets called when the MEX-file is cleared. See mexLock to see how to lock your MEX-file so that it is never cleared.

---

**See Also** mexAtExit, mexLock, mexMakeMemoryPersistent, and the mxCreate\* functions



# mexPrintf (C and Fortran)

---

<b>Purpose</b>	ANSI C printf-style output routine
<b>C Syntax</b>	<pre>#include "mex.h" int mexPrintf(const char *message, ...);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mexPrintf(message) character*(*) message</pre>
<b>Arguments</b>	<p>message String to be displayed. In C, the string may include formatting conversion characters, such as those used with the ANSI C printf function.</p> <p>... In C, any additional arguments needed to translate formatting conversion characters used in message. Each conversion character in message is converted to one of these values.</p>
<b>Returns</b>	The number of characters printed. This includes characters specified with backslash codes, such as \n and \b.
<b>Description</b>	<p>This routine prints a string on the screen and in the diary (if the diary is in use). It provides a callback to the standard C printf routine already linked inside MATLAB, and avoids linking the entire stdio library into your MEX-file.</p> <p>In a C MEX-file, you must call mexPrintf instead of printf to display a string.</p>

---

**Note** If you want the literal % in your message, you must use %% in your message string since % has special meaning to mexPrintf. Failing to do so causes unpredictable results.

---

## **C Examples**

See

- `mexfunction.c` in the `mex` subdirectory of the `examples` directory
- `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

## **See Also**

`mexErrMsgIdAndTxt`, `mexErrMsgTxt`, `mexWarnMsgIdAndTxt`,  
`mexWarnMsgTxt`

# mexPutVariable (C and Fortran)

---

<b>Purpose</b>	Copy mxArray from MEX-function into specified workspace						
<b>C Syntax</b>	<pre>#include "mex.h" int mexPutVariable(const char *workspace, const char *varname,     const mxArray *pm);</pre>						
<b>Fortran Syntax</b>	<pre>integer*4 function mexPutVariable(workspace, varname, pm) character*(*) workspace, varname mwPointer pm</pre>						
<b>Arguments</b>	<p>workspace Specifies the scope of the array that you are copying. The possible values are</p> <table><tr><td>base</td><td>Copy mxArray to the base workspace.</td></tr><tr><td>caller</td><td>Copy mxArray to the caller's workspace.</td></tr><tr><td>global</td><td>Copy mxArray to the list of global variables.</td></tr></table> <p>varname Name given to the mxArray in the workspace</p> <p>pm Pointer to the mxArray</p>	base	Copy mxArray to the base workspace.	caller	Copy mxArray to the caller's workspace.	global	Copy mxArray to the list of global variables.
base	Copy mxArray to the base workspace.						
caller	Copy mxArray to the caller's workspace.						
global	Copy mxArray to the list of global variables.						
<b>Returns</b>	0 on success; 1 on failure. A possible cause of failure is that pm is NULL in C (0 in Fortran).						
<b>Description</b>	<p>Call mexPutVariable to copy the mxArray, at pointer pm, from your MEX-function into the specified workspace. MATLAB gives the name, varname, to the copied mxArray in the receiving workspace.</p> <p>mexPutVariable makes the array accessible to other entities, such as MATLAB, M-files, or other MEX-functions.</p> <p>If a variable of the same name already exists in the specified workspace, mexPutVariable overwrites the previous contents of the variable with</p>						

the contents of the new mxArray. For example, suppose the MATLAB workspace defines variable Peaches as

```
Peaches
1      2      3      4
```

and you call mexPutVariable to copy Peaches into the same workspace:

```
mexPutVariable("base", "Peaches", pm)
```

Then the old value of Peaches disappears and is replaced by the value passed in by mexPutVariable.

## **C Examples**

See mexgetarray.c in the mex subdirectory of the examples directory.

## **See Also**

mexGetVariable

# mexSet (C)

---

**Purpose** Set value of specified Handle Graphics property

**C Syntax**

```
#include "mex.h"
int mexSet(double handle, const char *property,
           mxArray *value);
```

**Arguments**

handle  
Handle to a particular graphics object

property  
String naming a Handle Graphics property

value  
Pointer to an mxArray holding the new value to assign to the property

**Returns** 0 on success; 1 on failure. Possible causes of failure include:

- Specifying a nonexistent property.
- Specifying an illegal value for that property, for example, specifying a string value for a numerical property.

**Description** Call mexSet to set the value of the property of a certain graphics object. mexSet is the API equivalent of the MATLAB set function. To get the value of a graphics property, call mexGet.

**Examples** See mexget.c in the mex subdirectory of the examples directory.

**See Also** mexGet



**Purpose** Control response of mexCallMATLAB to errors

**C Syntax**

```
#include "mex.h"
void mexSetTrapFlag(int trapflag);
```

**Fortran Syntax**

```
subroutine mexSetTrapFlag(trapflag)
integer*4 trapflag
```

**Arguments**

trapflag  
Control flag. Possible values are

0	On error, control returns to the MATLAB prompt.
1	On error, control returns to your MEX-file.

**Description** Call mexSetTrapFlag to control the MATLAB response to errors in mexCallMATLAB.

If you do not call mexSetTrapFlag, then whenever MATLAB detects an error in a call to mexCallMATLAB, MATLAB automatically terminates the MEX-file and returns control to the MATLAB prompt. Calling mexSetTrapFlag with trapflag set to 0 is equivalent to not calling mexSetTrapFlag at all.

If you call mexSetTrapFlag and set the trapflag to 1, then whenever MATLAB detects an error in a call to mexCallMATLAB, MATLAB does not automatically terminate the MEX-file. Rather, MATLAB returns control to the line in the MEX-file immediately following the call to mexCallMATLAB. The MEX-file is then responsible for taking an appropriate response to the error.

If you call mexSetTrapFlag, the value of the trapflag you set remains in effect until the next call to mexSetTrapFlag within that MEX-file or, if there are no more calls to mexSetTrapFlag, until the MEX-file exits. If a routine defined in a MEX-file calls another MEX-file,

- 1 The current value of the trapflag in the first MEX-file is saved.

## mexSetTrapFlag (C and Fortran)

---

- 2 The second MEX-file is called with the `trapflag` initialized to 0 within that file.
- 3 When the second MEX-file exits, the saved value of the `trapflag` in the first MEX-file is restored within that file.

### **C Examples**

See `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory.

### **See Also**

`mexAtExit`, `mexErrMsgTxt`

**Purpose** Allow MEX-file to be cleared from memory

**C Syntax**

```
#include "mex.h"
void mexUnlock(void);
```

**Fortran  
Syntax**

```
subroutine mexUnlock()
```

**Description** By default, MEX-files are unlocked, meaning that a user can clear them at any time. Calling `mexLock` locks a MEX-file so that it cannot be cleared. Calling `mexUnlock` removes the lock so that the MEX-file can be cleared.

`mexLock` increments a lock count. If you called `mexLock` `n` times, you must call `mexUnlock` `n` times to unlock your MEX-file.

**C  
Examples** See `mexlock.c` in the `mex` subdirectory of the `examples` directory.

**See Also** `mexIsLocked`, `mexLock`, `mexMakeArrayPersistent`,  
`mexMakeMemoryPersistent`

# mexWarnMsgIdAndTxt (C and Fortran)

---

<b>Purpose</b>	Issue warning message with identifier
<b>C Syntax</b>	<pre>#include "mex.h" void mexWarnMsgIdAndTxt(const char *warningid,     const char *warningmsg, ...);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mexWarnMsgIdAndTxt(warningid, warningmsg) character*(*) warningid, warningmsg</pre>
<b>Arguments</b>	<p><code>warningid</code> String containing a MATLAB message identifier. See “Message Identifiers” in the MATLAB documentation for information on this topic.</p> <p><code>warningmsg</code> String containing the warning message to be displayed. In C, the string may include formatting conversion characters, such as those used with the ANSI C <code>sprintf</code> function.</p> <p>...</p> <p>In C, any additional arguments needed to translate formatting conversion characters used in <code>warningmsg</code>. Each conversion character in <code>warningmsg</code> is converted to one of these values.</p>
<b>Description</b>	<p>Call <code>mexWarnMsgIdAndTxt</code> to write a warning message and its corresponding identifier to the MATLAB window.</p> <p>Unlike <code>mexErrMsgIdAndTxt</code>, <code>mexWarnMsgIdAndTxt</code> does not cause the MEX-file to terminate.</p>
<b>See Also</b>	<code>mexErrMsgTxt</code> , <code>mexErrMsgIdAndTxt</code> , <code>mexWarnMsgTxt</code>

# mexWarnMsgTxt (C and Fortran)

---

<b>Purpose</b>	Issue warning message
<b>C Syntax</b>	<pre>#include "mex.h" void mexWarnMsgTxt(const char *warningmsg);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mexWarnMsgTxt(warningmsg) character*(*) warningmsg</pre>
<b>Arguments</b>	<p>warningmsg String containing the warning message to be displayed</p>
<b>Description</b>	<p>mexWarnMsgTxt causes MATLAB to display the contents of warningmsg. Unlike mexErrMsgTxt, mexWarnMsgTxt does not cause the MEX-file to terminate.</p>
<b>C Examples</b>	<p>See yprime.c in the mex subdirectory of the examples directory.</p> <p>Additional examples:</p> <ul style="list-style-type: none"><li>• explore.c in the mex subdirectory of the examples directory</li><li>• fulltosparse.c and revord.c in the refbook subdirectory of the examples directory</li><li>• mxisfinite.c and mxsetnzmax.c in the mx subdirectory of the examples directory</li></ul>
<b>See Also</b>	mexErrMsgTxt, mexErrMsgIdAndTxt, mexWarnMsgIdAndTxt

## mwIndex (C and Fortran)

---

**Purpose**           Type for index values

**C Syntax**           `#include "matrix.h"`

**Fortran  
Syntax**           `#include "fintrf.h"`

**Description**       mwIndex is a type that represents index values, such as indices into arrays. This function is provided for purposes of cross-platform flexibility. mwIndex is equivalent to `int` in C and `INTEGER*4` in Fortran.

In Fortran, mwIndex is implemented as a preprocessor macro.

**See Also**           `mex`, `mwSize`

**Purpose** Declare appropriate pointer type for platform

**Fortran Syntax** `#include "fintrf.h"`

**Description** `mwPointer` is a preprocessor macro that declares the appropriate Fortran type representing a pointer to an `mxAarray` or to other data that is not of a native Fortran type, such as memory allocated by `mxFmalloc`. On 32-bit platforms, the Fortran type that represents a pointer is `INTEGER*4`; on 64-bit platforms, it is `INTEGER*8`. The Fortran preprocessor translates `mwPointer` to the Fortran declaration that is appropriate for the platform on which you compile your file.

If your Fortran compiler supports preprocessing, you can use `mwPointer` to declare functions, arguments, and variables that represent pointers. If you cannot use `mwPointer`, you must ensure that your declarations have the correct size for the platform on which you are compiling Fortran code.

**Examples** This example declares the arguments for `mexFunction` in a Fortran MEX-file:

```
SUBROUTINE MEXFUNCTION(NLHS, PLHS, NRHS, PRHS)
  MWPOINTER PLHS(*), PRHS(*)
  INTEGER NLHS, NRHS
```

For additional examples, see the Fortran files with names ending in `.F` in the `$MATLAB/extern/examples` directory, where `$MATLAB` is the string returned by the `matlabroot` command.

## mwSize (C and Fortran)

---

<b>Purpose</b>	Type for size values
<b>C Syntax</b>	<code>#include "matrix.h"</code>
<b>Fortran Syntax</b>	<code>#include "fintrf.h"</code>
<b>Description</b>	<p>mwSize is a type that represents size values, such as array dimensions. This function is provided for purposes of cross-platform flexibility. mwSize is equivalent to int in C and INTEGER*4 in Fortran.</p> <p>In Fortran, mwSize is implemented as a preprocessor macro.</p>
<b>See Also</b>	mex, mwIndex



<b>Purpose</b>	Add field to structure array
<b>C Syntax</b>	<pre>#include "matrix.h" extern int mxAddField(mxArray pm, const char *fieldname);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxAddField(pm, fieldname) mwPointer pm character*(*) fieldname</pre>
<b>Arguments</b>	<p>pm     Pointer to a structure mxArray</p> <p>fieldname     The name of the field you want to add</p>
<b>Returns</b>	Field number on success or -1 if inputs are invalid or an out-of-memory condition occurs.
<b>Description</b>	Call <code>mxAddField</code> to add a field to a structure array. You must then create the values with the <code>mxCreate*</code> functions and use <code>mxSetFieldByNumber</code> to set the individual values for the field.
<b>See Also</b>	<code>mxRemoveField</code> , <code>mxSetFieldByNumber</code>

# mxArrayToString (C)

---

<b>Purpose</b>	Convert array to string
<b>C Syntax</b>	<pre>#include "matrix.h" char *mxArrayToString(const mxArray *array_ptr);</pre>
<b>Arguments</b>	<p>array_ptr Pointer to a string mxArray; that is, a pointer to an mxArray having the mxCHAR_CLASS class.</p>
<b>Returns</b>	A C-style string. Returns NULL on out of memory.
<b>Description</b>	<p>Call mxArrayToString to copy the character data of a string mxArray into a C-style string. The C-style string is always terminated with a NULL character.</p> <p>If the string array contains several rows, they are copied, one column at a time, into one long string array. This function is similar to mxGetString, except that</p> <ul style="list-style-type: none"><li>• It does not require the length of the string as an input.</li><li>• It supports multibyte character sets.</li></ul> <p>mxArrayToString does not free the dynamic memory that the char pointer points to. Consequently, you should typically free the string (using mxFree) immediately after you have finished using it.</p>
<b>Examples</b>	<p>See mexatexit.c in the mex subdirectory of the examples directory.</p> <p>For additional examples, see mxcreatecharmatrixfromstr.c and mxislogical.c in the mx subdirectory of the examples directory.</p>
<b>See Also</b>	<p>mxCreateCharArray, mxCreateCharMatrixFromStrings, mxCreateString, mxGetString</p>

<b>Purpose</b>	Check assertion value for debugging purposes
<b>C Syntax</b>	<pre>#include "matrix.h" void mxAssert(int expr, char *error_message);</pre>
<b>Arguments</b>	<p><code>expr</code> Value of assertion</p> <p><code>error_message</code> Description of why assertion failed</p>
<b>Description</b>	<p>Similar to the ANSI C <code>assert()</code> macro, <code>mxAssert</code> checks the value of an assertion, and continues execution only if the assertion holds. If <code>expr</code> evaluates to logical 1 (true), <code>mxAssert</code> does nothing. If <code>expr</code> evaluates to logical 0 (false), <code>mxAssert</code> prints an error to the MATLAB command window consisting of the failed assertion's expression, the filename and line number where the failed assertion occurred, and the <code>error_message</code> string. The <code>error_message</code> string allows you to specify a better description of why the assertion failed. Use an empty string if you don't want a description to follow the failed assertion message.</p> <p>After a failed assertion, control returns to the MATLAB command line.</p> <p>Note that the MEX script turns off these assertions when building optimized MEX-functions, so use this for debugging purposes only. Build the MEX-file using the syntax <code>mex -g filename</code> in order to use <code>mxAssert</code>.</p> <p>Assertions are a way of maintaining internal consistency of logic. Use them to keep yourself from misusing your own code and to prevent logical errors from propagating before they are caught; do not use assertions to prevent users of your code from misusing it.</p> <p>Assertions can be taken out of your code by the C preprocessor. You can use these checks during development and then remove them when the code works properly, letting you use them for troubleshooting during development without slowing down the final product.</p>

# mxAssertS (C)

---

**Purpose** Check assertion value without printing assertion text

**C Syntax**

```
#include "matrix.h"
void mxAssertS(int expr, char *error_message);
```

**Arguments**

expr  
Value of assertion

error\_message  
Description of why assertion failed

**Description** mxAssertS is similar to mxAssert, except mxAssertS does not print the text of the failed assertion. mxAssertS checks the value of an assertion, and continues execution only if the assertion holds. If expr evaluates to logical 1 (true), mxAssertS does nothing. If expr evaluates to logical 0 (false), mxAssertS prints an error to the MATLAB command window consisting of the filename and line number where the assertion failed and the error\_message string. The error\_message string allows you to specify a better description of why the assertion failed. Use an empty string if you don't want a description to follow the failed assertion message.

After a failed assertion, control returns to the MATLAB command line.

Note that the mex script turns off these assertions when building optimized MEX-functions, so use this for debugging purposes only. Build the MEX-file using the syntax `mex -g filename` in order to use mxAssert.

# mxCalcSingleSubscript (C and Fortran)

---

## Purpose

Offset from first element to desired element

## C Syntax

```
#include <matrix.h>
mwIndex mxCalcSingleSubscript(const mxArray *pm, mwSize nsubs,
                               mwIndex *subs);
```

## Fortran Syntax

```
mwIndex function mxCalcSingleSubscript(pm, nsubs, subs)
mwPointer pm
mwSize nsubs
mwIndex subs
```

## Arguments

pm

Pointer to an mxArray

nsubs

The number of elements in the subs array. Typically, you set nsubs equal to the number of dimensions in the mxArray that pm points to.

subs

An array of integers. Each value in the array should specify that dimension's subscript. In C syntax, the value in subs[0] specifies the row subscript, and the value in subs[1] specifies the column subscript. Use zero-based indexing for subscripts. For example, to express the starting element of a two-dimensional mxArray in subs, set subs[0] to 0 and subs[1] to 0.

In Fortran syntax, the value in subs(1) specifies the row subscript, and the value in subs(2) specifies the column subscript. Use 1-based indexing for subscripts. For example, to express the starting element of a two-dimensional mxArray in subs, set subs(1) to 1 and subs(2) to 1.

## Returns

The number of elements between the start of the mxArray and the specified subscript. This returned number is called an *index*; many mx routines (for example, mxGetField) require an index as an argument.

## mxCalcSingleSubscript (C and Fortran)

---

If `subs` describes the starting element of an `mxAarray`, `mxCalcSingleSubscript` returns 0. If `subs` describes the final element of an `mxAarray`, `mxCalcSingleSubscript` returns `N-1` (where `N` is the total number of elements).

### Description

Call `mxCalcSingleSubscript` to determine how many elements there are between the beginning of the `mxAarray` and a given element of that `mxAarray`. For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the first element of the array to the `(5,7)` element. Remember that the `mxAarray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxAarray` appears to have.

MATLAB uses a column-major numbering scheme to represent data elements internally. That means that MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on through the last column. For example, suppose you create a 4-by-2 variable. It is helpful to visualize the data as follows.

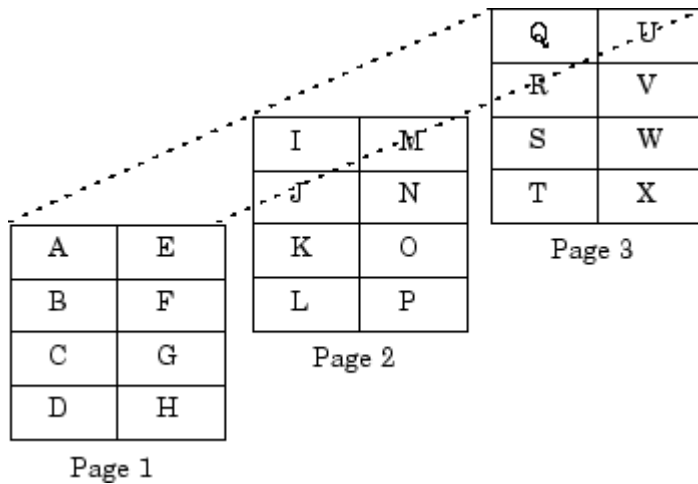
A	E
B	F
C	G
D	H

In fact, though, MATLAB internally represents the data as the following:

A	B	C	D	E	F	G	H
Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6	Index 7

If an `mxAarray` is `N`-dimensional, MATLAB represents the data in `N`-major order. For example, consider a three-dimensional array having dimensions 4-by-2-by-3. Although you can visualize the data as

# mxCalcSingleSubscript (C and Fortran)



MATLAB internally represents the data for this three-dimensional array in the following order:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Avoid using `mxCalcSingleSubscript` to traverse the elements of an array. In C, it is more efficient to do this by finding the array's starting address and then using pointer auto-incrementing to access successive elements. For example, to find the starting address of a numerical array, call `mxGetPr` or `mxGetPi`.

## C Examples

See `mxcalsinglesubscript.c` in the `mx` subdirectory of the examples directory.

## See Also

`mxGetCell`, `mxSetCell`

# mxCalloc (C and Fortran)

---

**Purpose** Allocate dynamic memory for array using MATLAB memory manager

**C Syntax**

```
#include "matrix.h"
#include <stdlib.h>
void *mxCalloc(size_t n, size_t size);
```

**Fortran Syntax**

```
mwPointer function mxCalloc(n, size)
mwSize n, size
```

**Arguments**

**n** Number of elements to allocate. This must be a nonnegative number.

**size** Number of bytes per element. (The C sizeof operator calculates the number of bytes per element.)

**Returns** A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, `mxCalloc` returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.

`mxCalloc` is unsuccessful when there is insufficient free heap space.

**Description** MATLAB applications should always call `mxCalloc` rather than `calloc` to allocate memory. Note that `mxCalloc` works differently in MEX-files than in stand-alone MATLAB applications.

In MEX-files, `mxCalloc` automatically

- Allocates enough contiguous heap space to hold `n` elements.
- Initializes all `n` elements to 0.
- Registers the returned heap space with the MATLAB memory management facility.



The MATLAB memory management facility maintains a list of all memory allocated by `mxCalloc`. The MATLAB memory management facility automatically frees (deallocates) all of a MEX-file's parcels when control returns to the MATLAB prompt.

In stand-alone MATLAB C applications, `mxCalloc` calls the ANSI C `calloc` function.

By default, in a MEX-file, `mxCalloc` generates nonpersistent `mxCalloc` data. In other words, the memory management facility automatically deallocates the memory as soon as the MEX-file ends. If you want the memory to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxCalloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX-file is cleared.

When you finish using the memory allocated by `mxCalloc`, call `mxFree`. `mxFree` deallocates the memory.

## C Examples

See

- `explore.c` in the `mex` subdirectory of the `examples` directory
- `phonebook.c` and `revord.c` in the `refbook` subdirectory of the `examples` directory

For additional examples, see `mxcalcsinglesubscript.c` and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxDestroyArray`, `mxFree`, `mxCalloc`, `mxCRealloc`

# mxChar (C)

---

<b>Purpose</b>	Data type for string mxArray
<b>C Syntax</b>	<code>typedef Uint16 mxChar;</code>
<b>Description</b>	All string mxArrays store their data elements as mxChar rather than as char. The MATLAB API defines an mxChar as a 16-bit unsigned integer.
<b>Examples</b>	See <code>mxmalloc.c</code> in the <code>mx</code> subdirectory of the <code>examples</code> directory. Additional examples: <ul style="list-style-type: none"><li>• <code>explore.c</code> in the <code>mex</code> subdirectory of the <code>examples</code> directory</li><li>• <code>mxcreatecharmatrixfromstr.c</code> in the <code>mx</code> subdirectory of the <code>examples</code> directory</li></ul>
<b>See Also</b>	<code>mxCreateCharArray</code>

**Purpose** Integer value identifying class of mxArray

**C Syntax**

```
typedef enum {
    mxUNKNOWN_CLASS = 0,
    mxCELL_CLASS,
    mxSTRUCT_CLASS,
    mxLOGICAL_CLASS,
    mxCHAR_CLASS,
    <unused>,
    mxDOUBLE_CLASS,
    mxSINGLE_CLASS,
    mxINT8_CLASS,
    mxUINT8_CLASS,
    mxINT16_CLASS,
    mxUINT16_CLASS,
    mxINT32_CLASS,
    mxUINT32_CLASS,
    mxINT64_CLASS,
    mxUINT64_CLASS,
    mxFUNCTION_CLASS
} mxClassID;
```

**Constants**

**mxUNKNOWN\_CLASS**  
The class cannot be determined. You cannot specify this category for an mxArray; however, mxGetClassID can return this value if it cannot identify the class.

**mxCELL\_CLASS**  
Identifies a cell mxArray.

**mxSTRUCT\_CLASS**  
Identifies a structure mxArray.

**mxLOGICAL\_CLASS**  
Identifies a logical mxArray, an mxArray that stores Boolean elements logical 1 (true) and logical 0 (false).

## mxClassID (C)

---

- `mxCHAR_CLASS`  
Identifies a string mxArray, an mxArray whose data is represented as mxCHAR's.
- `mxDOUBLE_CLASS`  
Identifies a numeric mxArray whose data is stored as double-precision, floating-point numbers.
- `mxSINGLE_CLASS`  
Identifies a numeric mxArray whose data is stored as single-precision, floating-point numbers.
- `mxINT8_CLASS`  
Identifies a numeric mxArray whose data is stored as signed 8-bit integers.
- `mxUINT8_CLASS`  
Identifies a numeric mxArray whose data is stored as unsigned 8-bit integers.
- `mxINT16_CLASS`  
Identifies a numeric mxArray whose data is stored as signed 16-bit integers.
- `mxUINT16_CLASS`  
Identifies a numeric mxArray whose data is stored as unsigned 16-bit integers.
- `mxINT32_CLASS`  
Identifies a numeric mxArray whose data is stored as signed 32-bit integers.
- `mxUINT32_CLASS`  
Identifies a numeric mxArray whose data is stored as unsigned 32-bit integers.
- `mxINT64_CLASS`  
Identifies a numeric mxArray whose data is stored as signed 64-bit integers.

`mxUINT64_CLASS`

Identifies a numeric `mxArray` whose data is stored as unsigned 64-bit integers.

`mxFUNCTION_CLASS`

Identifies a function handle `mxArray`.

### **Description**

Various `mx*` calls require or return an `mxClassID` argument. `mxClassID` identifies the way in which the `mxArray` represents its data elements.

### **Examples**

See `explore.c` in the `mex` subdirectory of the `examples` directory.

### **See Also**

`mxCreateNumericArray`

# mxClassIDFromClassName (Fortran)

---

**Purpose** Identifier corresponding to class

**Fortran Syntax** integer\*4 function mxClassIDFromClassName(classname)  
character\*(\*) classname

**Arguments** classname  
A character array specifying a MATLAB class name. Use one of the strings from the following table.

**Returns** A numeric identifier used internally by MATLAB to represent the MATLAB class, classname. Returns 0 if classname is not a recognized MATLAB class.

**Description** Use mxClassIDFromClassName to obtain an identifier for any class that is recognized by MATLAB. This function is most commonly used to provide a classid argument to mxCreateNumericArray and mxCreateNumericMatrix.

Valid choices for classname are listed in the following table. MATLAB returns 0 if classname is unrecognized.

cell	char	double	function_handle
int8	int16	int32	logical
object	single	struct	uint8
uint16	uint32		

**See Also** mxGetClassName, mxCreateNumericArray, mxCreateNumericMatrix

<b>Purpose</b>	Flag specifying whether mxArray has imaginary components
<b>C Syntax</b>	<pre>typedef enum mxComplexity {mxREAL=0, mxCOMPLEX};</pre>
<b>Constants</b>	<p>mxREAL Identifies an mxArray with no imaginary components.</p> <p>mxCOMPLEX Identifies an mxArray with imaginary components.</p>
<b>Description</b>	Various mx* calls require an mxComplexity argument. You can set an mxComplex argument to either mxREAL or mxCOMPLEX.
<b>Examples</b>	See mxcalcsinglesubscript.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxCreateNumericArray, mxCreateDoubleMatrix, mxCreateSparse

# mxCopyCharacterToPtr (Fortran)

---

**Purpose** Copy character values from Fortran array to pointer array

**Fortran Syntax** subroutine mxCopyCharacterToPtr(y, px, n)  
character\*(\*) y  
mwPointer px  
mwSize n

**Arguments**

y  
character Fortran array

px  
Pointer to character or name array

n  
Number of elements to copy

**Description** mxCopyCharacterToPtr copies n character values from the Fortran character array y into the MATLAB string array pointed to by px. This subroutine is essential for copying character data between MATLAB pointer arrays and ordinary Fortran character arrays.

**See Also** mxCopyPtrToCharacter, mxCreateCharArray, mxCreateString, mxCreateCharMatrixFromStrings



# mxCopyComplex16ToPtr (Fortran)

---

**Purpose** Copy COMPLEX\*16 values from Fortran array to pointer array

**Fortran Syntax**

```
subroutine mxCopyComplex16ToPtr(y, pr, pi, n)
  complex*16 y(n)
  mwPointer pr, pi
  mwSize n
```

**Arguments**

`y`  
COMPLEX\*16 Fortran array

`pr`  
Pointer to the real data of a double-precision MATLAB array

`pi`  
Pointer to the imaginary data of a double-precision MATLAB array

`n`  
Number of elements to copy

**Description** `mxCopyComplex16ToPtr` copies `n` COMPLEX\*16 values from the Fortran COMPLEX\*16 array `y` into the MATLAB arrays pointed to by `pr` and `pi`. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

**See Also** `mxCopyPtrToComplex16`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

# mxCopyComplex8ToPtr (Fortran)

---

**Purpose** Copy COMPLEX\*8 values from Fortran array to pointer array

**Fortran Syntax**

```
subroutine mxCopyComplex8ToPtr(y, pr, pi, n)
  complex*8 y(n)
  mwPointer pr, pi
  mwSize n
```

**Arguments**

y  
COMPLEX\*8 Fortran array

pr  
Pointer to the real data of a single-precision MATLAB array

pi  
Pointer to the imaginary data of a single-precision MATLAB array

n  
Number of elements to copy

**Description** mxCopyComplex8ToPtr copies n COMPLEX\*8 values from the Fortran COMPLEX\*8 array y into the MATLAB arrays pointed to by pr and pi. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

**See Also** mxCopyPtrToComplex8, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

**Purpose** Copy INTEGER\*1 values from Fortran array to pointer array

**Fortran Syntax**

```
subroutine mxCopyInteger1ToPtr(y, px, n)
integer*1 y(n)
mwPointer px
mwSize n
```

**Arguments**

y  
INTEGER\*1 Fortran array

px  
Pointer to ir or jc array

n  
Number of elements to copy

**Description** mxCopyInteger1ToPtr copies n INTEGER\*1 values from the Fortran INTEGER\*1 array y into the MATLAB array pointed to by px, either an ir or jc array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

---

**Note** This function can only be used with sparse matrices.

---

**See Also** mxCopyPtrToInteger1, mxCreateNumericArray, mxCreateNumericMatrix

# mxCopyInteger2ToPtr (Fortran)

---

**Purpose** Copy INTEGER\*2 values from Fortran array to pointer array

**Fortran Syntax**

```
subroutine mxCopyInteger2ToPtr(y, px, n)
integer*2 y(n)
mwPointer px
mwSize n
```

**Arguments**

y  
INTEGER\*2 Fortran array

px  
Pointer to ir or jc array

n  
Number of elements to copy

**Description** mxCopyInteger2ToPtr copies n INTEGER\*2 values from the Fortran INTEGER\*2 array y into the MATLAB array pointed to by px, either an ir or jc array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

---

**Note** This function can only be used with sparse matrices.

---

**See Also** mxCopyPtrToInteger2, mxCreateNumericArray, mxCreateNumericMatrix

**Purpose** Copy INTEGER\*4 values from Fortran array to pointer array

**Fortran Syntax**

```
subroutine mxCopyInteger4ToPtr(y, px, n)
integer*4 y(n)
mwPointer px
mwSize n
```

**Arguments**

y  
INTEGER\*4 Fortran array

px  
Pointer to ir or jc array

n  
Number of elements to copy

**Description** mxCopyInteger4ToPtr copies n INTEGER\*4 values from the Fortran INTEGER\*4 array y into the MATLAB array pointed to by px, either an ir or jc array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

---

**Note** This function can only be used with sparse matrices.

---

**See Also** mxCopyPtrToInteger4, mxCreateNumericArray, mxCreateNumericMatrix

# mxCopyPtrToCharacter (Fortran)

---

**Purpose** Copy character values from pointer array to Fortran array

**Fortran Syntax** subroutine mxCopyPtrToCharacter(px, y, n)  
mwPointer px  
character\*(\*) y  
mwSize n

**Arguments** px Pointer to character or name array  
y character Fortran array  
n Number of elements to copy

**Description** mxCopyPtrToCharacter copies n character values from the MATLAB array pointed to by px into the Fortran character array y. This subroutine is essential for copying character data from MATLAB pointer arrays into ordinary Fortran character arrays.

**Examples** See matdemo2.F in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

**See Also** mxCopyCharacterToPtr, mxCreateCharArray, mxCreateString, mxCreateCharMatrixFromStrings

# mxCopyPtrToComplex16 (Fortran)

---

<b>Purpose</b>	Copy COMPLEX*16 values from pointer array to Fortran array
<b>Fortran Syntax</b>	<pre>subroutine mxCopyPtrToComplex16(pr, pi, y, n) mwPointer pr, pi complex*16 y(n) mwSize n</pre>
<b>Arguments</b>	<p><code>pr</code> Pointer to the real data of a double-precision MATLAB array</p> <p><code>pi</code> Pointer to the imaginary data of a double-precision MATLAB array</p> <p><code>y</code> COMPLEX*16 Fortran array</p> <p><code>n</code> Number of elements to copy</p>
<b>Description</b>	<p><code>mxCopyPtrToComplex16</code> copies <code>n</code> COMPLEX*16 values from the MATLAB arrays pointed to by <code>pr</code> and <code>pi</code> into the Fortran COMPLEX*16 array <code>y</code>. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.</p>
<b>See Also</b>	<code>mxCopyComplex16ToPtr</code> , <code>mxCreateNumericArray</code> , <code>mxCreateNumericMatrix</code> , <code>mxGetData</code> , <code>mxGetImagData</code>

# mxCopyPtrToComplex8 (Fortran)

---

<b>Purpose</b>	Copy COMPLEX*8 values from pointer array to Fortran array
<b>Fortran Syntax</b>	<pre>subroutine mxCopyPtrToComplex8(pr, pi, y, n) mwPointer pr, pi complex*8 y(n) mwSize n</pre>
<b>Arguments</b>	<p><code>pr</code> Pointer to the real data of a single-precision MATLAB array</p> <p><code>pi</code> Pointer to the imaginary data of a single-precision MATLAB array</p> <p><code>y</code> COMPLEX*8 Fortran array</p> <p><code>n</code> Number of elements to copy</p>
<b>Description</b>	<p><code>mxCopyPtrToComplex8</code> copies <code>n</code> COMPLEX*8 values from the MATLAB arrays pointed to by <code>pr</code> and <code>pi</code> into the Fortran COMPLEX*8 array <code>y</code>. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.</p>
<b>See Also</b>	<code>mxCopyComplex8ToPtr</code> , <code>mxCreateNumericArray</code> , <code>mxCreateNumericMatrix</code> , <code>mxGetData</code> , <code>mxGetImagData</code>



**Purpose** Copy INTEGER\*1 values from pointer array to Fortran array

**Fortran Syntax**

```
subroutine mxCopyPtrToInteger1(px, y, n)
mwPointer px
integer*1 y(n)
mwSize n
```

**Arguments**

px  
Pointer to ir or jc array

y  
INTEGER\*1 Fortran array

n  
Number of elements to copy

**Description** mxCopyPtrToInteger1 copies n INTEGER\*1 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER\*1 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

---

**Note** This function can only be used with sparse matrices.

---

**See Also** mxCopyInteger1ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

# mxCopyPtrToInteger2 (Fortran)

---

**Purpose** Copy INTEGER\*2 values from pointer array to Fortran array

**Fortran Syntax**

```
subroutine mxCopyPtrToInteger2(px, y, n)
mwPointer px
integer*2 y(n)
mwSize n
```

**Arguments**

px Pointer to ir or jc array

y INTEGER\*2 Fortran array

n Number of elements to copy

**Description** mxCopyPtrToInteger2 copies n INTEGER\*2 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER\*2 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

---

**Note** This function can only be used with sparse matrices.

---

**See Also** mxCopyInteger2ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

**Purpose** Copy INTEGER\*4 values from pointer array to Fortran array

**Fortran Syntax**

```
subroutine mxCopyPtrToInteger4(px, y, n)
mwPointer px
integer*4 y(n)
mwSize n
```

**Arguments**

px  
Pointer to ir or jc array

y  
INTEGER\*4 Fortran array

n  
Number of elements to copy

**Description** mxCopyPtrToInteger4 copies n INTEGER\*4 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER\*4 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

---

**Note** This function can only be used with sparse matrices.

---

**See Also** mxCopyInteger4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

# mxCopyPtrToPtrArray (Fortran)

---

**Purpose** Copy pointer values from pointer array to Fortran array

**Fortran Syntax**

```
subroutine mxCopyPtrToPtrArray(px, y, n)
mwPointer px
mwPointer y(n)
mwSize n
```

**Arguments**

px	Pointer to pointer array
y	Fortran array of mwPointer values
n	Number of pointers to copy

**Description** mxCopyPtrToPtrArray copies n pointers from the MATLAB array pointed to by px into the Fortran array y. This subroutine is essential for copying the output of matGetDir into an array of pointers. After calling this function, each element of y contains a pointer to a string. You can convert these strings to Fortran character arrays by passing each element of y as the first argument to mxCopyPtrToCharacter.

**Examples** See matdemo2.F in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

**See Also** matGetDir, mxCopyPtrToCharacter

<b>Purpose</b>	Copy REAL*4 values from pointer array to Fortran array
<b>Fortran Syntax</b>	<pre>subroutine mxCopyPtrToReal4(px, y, n) mwPointer px real*4 y(n) mwSize n</pre>
<b>Arguments</b>	<p>px Pointer to the real or imaginary data of a single-precision MATLAB array</p> <p>y REAL*4 Fortran array</p> <p>n Number of elements to copy</p>
<b>Description</b>	<p>mxCopyPtrToReal4 copies n REAL*4 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*4 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.</p>
<b>See Also</b>	<p>mxCopyReal4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData</p>

# mxCopyPtrToReal8 (Fortran)

---

<b>Purpose</b>	Copy REAL*8 values from pointer array to Fortran array
<b>Fortran Syntax</b>	<pre>subroutine mxCopyPtrToReal8(px, y, n) mwPointer px real*8 y(n) mwSize n</pre>
<b>Arguments</b>	<p>px Pointer to the real or imaginary data of a double-precision MATLAB array</p> <p>y REAL*8 Fortran array</p> <p>n Number of elements to copy</p>
<b>Description</b>	mxCopyPtrToReal8 copies n REAL*8 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*8 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.
<b>Examples</b>	See fengdemo.F in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.
<b>See Also</b>	mxCopyReal8ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

<b>Purpose</b>	Copy REAL*4 values from Fortran array to pointer array
<b>Fortran Syntax</b>	<pre>subroutine mxCopyReal4ToPtr(y, px, n) real*4 y(n) mwPointer px mwSize n</pre>
<b>Arguments</b>	<p>y REAL*4 Fortran array</p> <p>px Pointer to the real or imaginary data of a single-precision MATLAB array</p> <p>n Number of elements to copy</p>
<b>Description</b>	<p>mxCopyReal4ToPtr(y,px,n) copies n REAL*4 values from the Fortran REAL*4 array y into the MATLAB array pointed to by px, either a pr or pi array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.</p>
<b>See Also</b>	<p>mxCopyPtrToReal4, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData</p>

## mxCopyReal8ToPtr (Fortran)

---

**Purpose** Copy REAL\*8 values from Fortran array to pointer array

**Fortran Syntax**

```
subroutine mxCopyReal8ToPtr(y, px, n)
real*8 y(n)
mwPointer px
mwSize n
```

**Arguments**

y  
REAL\*8 Fortran array

px  
Pointer to the real or imaginary data of a double-precision MATLAB array

n  
Number of elements to copy

**Description** mxCopyReal8ToPtr(y,px,n) copies n REAL\*8 values from the Fortran REAL\*8 array y into the MATLAB array pointed to by px, either a pr or pi array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

**Examples** See matdemo1.F and fengdemo.F in the eng\_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

**See Also** mxCopyPtrToReal8, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData



# mxCreateCellArray (C and Fortran)

---

<b>Purpose</b>	Create unpopulated N-D cell mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateCellArray(mwSize ndim, const mwSize *dims);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateCellArray(ndim, dims) mwSize ndim, dims</pre>
<b>Arguments</b>	<p><code>ndim</code> The desired number of dimensions in the created cell. For example, to create a three-dimensional cell mxArray, set <code>ndim</code> to 3.</p> <p><code>dims</code> The dimensions array. Each element in the dimensions array contains the size of the mxArray in that dimension. For example, in C, setting <code>dims[0]</code> to 5 and <code>dims[1]</code> to 7 establishes a 5-by-7 mxArray. In Fortran, setting <code>dims(1)</code> to 5 and <code>dims(2)</code> to 7 establishes a 5-by-7 mxArray. In most cases, there should be <code>ndim</code> elements in the <code>dims</code> array.</p>
<b>Returns</b>	A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, <code>mxCreateCellArray</code> returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. The most common cause of failure is insufficient free heap space.
<b>Description</b>	<p>Use <code>mxCreateCellArray</code> to create a cell mxArray whose size is defined by <code>ndim</code> and <code>dims</code>. For example, in C, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set</p> <pre>ndim = 3; dims[0] = 4; dims[1] = 8; dims[2] = 7;</pre> <p>In Fortran, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set</p> <pre>ndim = 3;</pre>

## mxCreateCellArray (C and Fortran)

---

```
dims(1) = 4; dims(2) = 8; dims(3) = 7;
```

The created cell mxArray is unpopulated; mxCreateCellArray initializes each cell to NULL. To put data into a cell, call mxSetCell.

Any trailing singleton dimensions specified in the dims argument are automatically removed from the resulting array. For example, if ndim equals 5 and dims equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.

### **C Examples**

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

### **See Also**

`mxCreateCellMatrix`, `mxGetCell`, `mxSetCell`, `mxIsCell`

# mxCreateCellMatrix (C and Fortran)

---

<b>Purpose</b>	Create unpopulated 2-D cell mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateCellMatrix(mwSize m, mwSize n);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateCellMatrix(m, n) mwSize m, n</pre>
<b>Arguments</b>	<p>m The desired number of rows</p> <p>n The desired number of columns</p>
<b>Returns</b>	A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateCellMatrix returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for mxCreateCellMatrix to be unsuccessful.
<b>Description</b>	<p>Use mxCreateCellMatrix to create an m-by-n two-dimensional cell mxArray. The created cell mxArray is unpopulated; mxCreateCellMatrix initializes each cell to NULL in C (0 in Fortran). To put data into cells, call mxSetCell.</p> <p>mxCreateCellMatrix is identical to mxCreateCellArray except that mxCreateCellMatrix can create two-dimensional mxArrays only, but mxCreateCellArray can create mxArrays having any number of dimensions greater than 1.</p>
<b>C Examples</b>	See mxcreatecellmatrix.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxCreateCellArray

# mxCreateCharArray (C and Fortran)

---

<b>Purpose</b>	Create unpopulated N-D string mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateCharArray(mwSize ndim, const mwSize *dims);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateCharArray(ndim, dims) mwSize ndim, dims</pre>
<b>Arguments</b>	<p><code>ndim</code> The desired number of dimensions in the string mxArray. You must specify a positive number. If you specify 0, 1, or 2, mxCreateCharArray creates a two-dimensional mxArray.</p> <p><code>dims</code> The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting <code>dims[0]</code> to 5 and <code>dims[1]</code> to 7 establishes a 5-by-7 mxArray. In Fortran, setting <code>dims(1)</code> to 5 and <code>dims(2)</code> to 7 establishes a 5-by-7 character mxArray. The <code>dims</code> array must have at least <code>ndim</code> elements.</p>
<b>Returns</b>	A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateCharArray returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for mxCreateCharArray to be unsuccessful.
<b>Description</b>	<p>Call mxCreateCharArray to create an N-dimensional string mxArray. The created mxArray is unpopulated; that is, mxCreateCharArray initializes each cell to NULL in C (0 in Fortran).</p> <p>Any trailing singleton dimensions specified in the <code>dims</code> argument are automatically removed from the resulting array. For example, if <code>ndim</code> equals 5 and <code>dims</code> equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.</p>

# mxCreateCharArray (C and Fortran)

---

## **C Examples**

See `mxcreatecharmatrixfromstr.c` in the `mx` subdirectory of the `examples` directory.

## **See Also**

`mxCreateCharMatrixFromStrings`, `mxCreateString`

# mxCreateCharMatrixFromStrings (C and Fortran)

---

<b>Purpose</b>	Create populated 2-D string mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateCharMatrixFromStrings(mwSize m, const char **str);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateCharMatrixFromStrings(m, str) mwSize m character*(*) str(m)</pre>
<b>Arguments</b>	<p><b>m</b> The desired number of rows in the created string mxArray. The value you specify for <b>m</b> should equal the number of strings in <b>str</b>.</p> <p><b>str</b> In C, an array of strings containing at least <b>m</b> strings. In Fortran, a character*n array of size <b>m</b>, where each element of the array is <b>n</b> bytes.</p>
<b>Returns</b>	A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateCharMatrixFromStrings returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the primary reason for mxCreateCharArray to be unsuccessful. Another possible reason for failure is that <b>str</b> contains fewer than <b>m</b> strings.
<b>Description</b>	<p>Use mxCreateCharMatrixFromStrings to create a two-dimensional string mxArray, where each row is initialized to a string from <b>str</b>. In C, the created mxArray has dimensions <b>m-by-max</b>, where <b>max</b> is the length of the longest string in <b>str</b>. In Fortran, the created mxArray has dimensions <b>m-by-n</b>, where <b>n</b> is the number of characters in <b>str(i)</b>.</p> <p>Note that string mxArrays represent their data elements as mxChar rather than as C char.</p>
<b>C Examples</b>	See mxcreatecharmatrixfromstr.c in the mx subdirectory of the examples directory.

# mxCreateCharMatrixFromStrings (C and Fortran)

---

**See Also**      `mxCreateCharArray`, `mxCreateString`, `mxGetString`

# mxCreateDoubleMatrix (C and Fortran)

---

<b>Purpose</b>	Create unpopulated 2-D, double-precision, floating-point mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateDoubleMatrix(mwSize m, mwSize n,     mxComplexity ComplexFlag);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateDoubleMatrix(m, n, ComplexFlag) mwSize m, n integer*4 ComplexFlag</pre>
<b>Arguments</b>	<p>m The desired number of rows</p> <p>n The desired number of columns</p> <p>ComplexFlag Specify either mxREAL or mxCOMPLEX. If the data you plan to put into the mxArray has no imaginary components, specify mxREAL in C (0 in Fortran). If the data has some imaginary components, specify mxCOMPLEX in C (1 in Fortran).</p>
<b>Returns</b>	A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateDoubleMatrix returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateDoubleMatrix is unsuccessful when there is not enough free heap space to create the mxArray.
<b>Description</b>	<p>Use mxCreateDoubleMatrix to create an m-by-n mxArray. mxCreateDoubleMatrix initializes each element in the pr array to 0. If you set ComplexFlag to mxCOMPLEX in C (1 in Fortran), mxCreateDoubleMatrix also initializes each element in the pi array to 0.</p> <p>If you set ComplexFlag to mxREAL in C (0 in Fortran), mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements. If you set ComplexFlag to mxCOMPLEX in C (1 in Fortran),</p>



## mxCreateDoubleMatrix (C and Fortran)

---

`mxCreateDoubleMatrix` allocates enough memory to hold m-by-n real elements and m-by-n imaginary elements.

Call `mxDestroyArray` when you finish using the `mxArray`.

`mxDestroyArray` deallocates the `mxArray` and its associated real and complex elements.

### **C Examples**

See `convec.c`, `findnz.c`, `sincall.c`, `timestwo.c`, `timestwoalt.c`, and `xtimesy.c` in the `refbook` subdirectory of the `examples` directory.

### **See Also**

`mxCreateNumericArray`

# mxCreateDoubleScalar (C and Fortran)

---

**Purpose** Create scalar, double-precision array initialized to specified value

**C Syntax**

```
#include "matrix.h"
mxArray *mxCreateDoubleScalar(double value);
```

**Fortran Syntax**

```
mwPointer function mxCreateDoubleScalar(value)
real*8 value
```

**Arguments** value  
The desired value to which you want to initialize the array

**Returns** A pointer to the created mxArray, if successful. mxCreateDoubleScalar is unsuccessful if there is not enough free heap space to create the mxArray. If mxCreateDoubleScalar is unsuccessful in a MEX-file, the MEX-file prints an “Out of Memory” message, terminates, and control returns to the MATLAB prompt. If mxCreateDoubleScalar is unsuccessful in a stand-alone (nonMEX-file) application, mxCreateDoubleScalar returns NULL in C (0 in Fortran).

**Description** Call mxCreateDoubleScalar to create a scalar double mxArray. mxCreateDoubleScalar is a convenience function that can be used in place of the following C code:

```
pa = mxCreateDoubleMatrix(1, 1, mxREAL);
*mxGetPr(pa) = value;
```

mxCreateDoubleScalar can be used in place of the following Fortran code:

```
pm = mxCreateDoubleMatrix(1, 1, 0)
mxCopyReal8ToPtr(value, mxGetPr(pm), 1)
```

When you finish using the mxArray, call mxDestroyArray to destroy it.

**See Also** mxGetPr, mxCreateDoubleMatrix

<b>Purpose</b>	Create N-D logical mxArray initialized to false
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateLogicalArray(mwSize ndim, const mwSize *dims);</pre>
<b>Arguments</b>	<p><b>ndim</b> Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateLogicalArray automatically sets the number of dimensions to 2.</p> <p><b>dims</b> The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxArray. There should be ndim elements in the dims array.</p>
<b>Returns</b>	A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateLogicalArray returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateLogicalArray is unsuccessful when there is not enough free heap space to create the mxArray.
<b>Description</b>	<p>Call mxCreateLogicalArray to create an N-dimensional mxArray of logical 1 (true) and logical 0 (false) elements. After creating the mxArray, mxCreateLogicalArray initializes all its elements to logical 0. mxCreateLogicalArray differs from mxCreateLogicalMatrix in that the latter can create two-dimensional arrays only.</p> <p>mxCreateLogicalArray allocates dynamic memory to store the created mxArray. When you finish with the created mxArray, call mxDestroyArray to deallocate its memory.</p> <p>Any trailing singleton dimensions specified in the dims argument are automatically removed from the resulting array. For example, if ndim equals 5 and dims equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.</p>

## mxCreateLogicalArray (C)

---

### **See Also**

`mxCreateLogicalMatrix`, `mxCreateSparseLogicalMatrix`,  
`mxCreateLogicalScalar`

<b>Purpose</b>	Create 2-D, logical mxArray initialized to false
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateLogicalMatrix(mwSize m, mwSize n);</pre>
<b>Arguments</b>	<p>m The desired number of rows</p> <p>n The desired number of columns</p>
<b>Returns</b>	A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateLogicalMatrix returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateLogicalMatrix is unsuccessful when there is not enough free heap space to create the mxArray.
<b>Description</b>	<p>Use mxCreateLogicalMatrix to create an m-by-n mxArray of logical 1 (true) and logical 0 (false) elements. mxCreateLogicalMatrix initializes each element in the array to logical 0.</p> <p>Call mxDestroyArray when you finish using the mxArray. mxDestroyArray deallocates the mxArray.</p>
<b>See Also</b>	mxCreateLogicalArray, mxCreateSparseLogicalMatrix, mxCreateLogicalScalar

# mxCreateLogicalScalar (C)

---

**Purpose** Create scalar, logical mxArray initialized to false

**C Syntax**

```
#include "matrix.h"
mxArray *mxCreateLogicalScalar(mxLogical value);
```

**Arguments** value  
The desired logical value, logical 1 (true) or logical 0 (false), to which you want to initialize the array

**Returns** A pointer to the created mxArray, if successful. mxCreateLogicalScalar is unsuccessful if there is not enough free heap space to create the mxArray. If mxCreateLogicalScalar is unsuccessful in a MEX-file, the MEX-file prints an “Out of Memory” message, terminates, and control returns to the MATLAB prompt. If mxCreateLogicalScalar is unsuccessful in a stand-alone (non-MEX-file) application, the function returns NULL.

**Description** Call mxCreateLogicalScalar to create a scalar logical mxArray. mxCreateLogicalScalar is a convenience function that can be used in place of the following code:

```
pa = mxCreateLogicalMatrix(1, 1);
*mxGetLogicals(pa) = value;
```

When you finish using the mxArray, call mxDestroyArray to destroy it.

**See Also** mxCreateLogicalArray, mxCreateLogicalMatrix, mxIsLogicalScalar, mxIsLogicalScalarTrue, mxGetLogicals

# mxCreateNumericArray (C and Fortran)

---

<b>Purpose</b>	Create unpopulated N-D numeric mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateNumericArray(mwSize ndim, const mwSize *dims,                                mxClassID classid, mxComplexity ComplexFlag);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateNumericArray(ndim, dims, classid,  ComplexFlag) mwSize ndim, dims integer*4 classid, ComplexFlag</pre>
<b>Arguments</b>	<p><b>ndim</b> Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateNumericArray automatically sets the number of dimensions to 2.</p> <p><b>dims</b> The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxArray. In Fortran, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 mxArray. In most cases, there should be ndim elements in the dims array.</p> <p><b>classid</b> An identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying mxINT16_CLASS in C causes each piece of numerical data in the mxArray to be represented as a 16-bit signed integer. In Fortran, use the function mxClassIDFromClassname to derive the classid value from a MATLAB class name. See the Description section for more information.</p> <p><b>ComplexFlag</b> If the data you plan to put into the mxArray has no imaginary components, specify mxREAL in C (0 in Fortran). If the data has some imaginary components, specify mxCOMPLEX in C (1 in Fortran).</p>

# mxCreateNumericArray (C and Fortran)

---

## Returns

A pointer to the created `mxArray`, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, `mxCreateNumericArray` returns `NULL` in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. `mxCreateNumericArray` is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Call `mxCreateNumericArray` to create an N-dimensional `mxArray` in which all data elements have the numeric data type specified by `classid`. After creating the `mxArray`, `mxCreateNumericArray` initializes all its real data elements to 0. If `ComplexFlag` equals `mxCOMPLEX` in C (1 in Fortran), `mxCreateNumericArray` also initializes all its imaginary data elements to 0. `mxCreateNumericArray` differs from `mxCreateDoubleMatrix` in two important respects:

- All data elements in `mxCreateDoubleMatrix` are double-precision, floating-point numbers. The data elements in `mxCreateNumericArray` could be any numerical type, including different integer precisions.
- `mxCreateDoubleMatrix` can create two-dimensional arrays only; `mxCreateNumericArray` can create arrays of two or more dimensions.

`mxCreateNumericArray` allocates dynamic memory to store the created `mxArray`. When you finish with the created `mxArray`, call `mxDestroyArray` to deallocate its memory.

Any trailing singleton dimensions specified in the `dims` argument are automatically removed from the resulting array. For example, if `ndim` equals 5 and `dims` equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.

The following table shows the C `classid` values and the Fortran data types that are equivalent to MATLAB classes.



# mxCreateNumericArray (C and Fortran)

<b>MATLAB Class Name</b>	<b>C classid Value</b>	<b>Fortran Type</b>
int8	mxINT8_CLASS	BYTE
uint8	mxUINT8_CLASS	
int16	mxUINT16_CLASS	INTEGER*2
uint16	mxUINT16_CLASS	
int32	mxINT32_CLASS	INTEGER*4
uint32	mxUINT32_CLASS	
int64	mxINT64_CLASS	INTEGER*8
uint64	mxUINT64_CLASS	
single	mxSINGLE_CLASS	REAL*4
double	mxDOUBLE_CLASS	REAL*8
single, with imaginary components	mxSINGLE_CLASS	COMPLEX*8
double, with imaginary components	mxDOUBLE_CLASS	COMPLEX*16

## **C Examples**

See `phonebook.c` and `doubleelement.c` in the `refbook` subdirectory of the `examples` directory. For an additional example, see `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

## **Fortran Examples**

To create a 4-by-4-by-2 array of REAL\*8 elements having no imaginary components, use

```
C      Create 4x4x2 mxArray of REAL*8
      data dims / 4, 4, 2 /
      mxCreateNumericArray(3, dims,
+                          mxClassIDFromClassName('double'), 0)
```

## mxCreateNumericArray (C and Fortran)

---

### **See Also**

`mxClassId`, `mxClassIdFromClassName`, `mxComplexity`,  
`mxCreateNumericMatrix`

# mxCreateNumericMatrix (C and Fortran)

---

<b>Purpose</b>	Create numeric matrix and initialize data elements to 0
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateNumericMatrix(mwSize m, mwSize n,     mxClassID classid, mxComplexity ComplexFlag);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateNumericMatrix(m, n, classid,     ComplexFlag) mwSize m, n integer*4 classid, ComplexFlag</pre>
<b>Arguments</b>	<p><b>m</b> The desired number of rows.</p> <p><b>n</b> The desired number of columns.</p> <p><b>classid</b> An identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying <code>mxINT16_CLASS</code> in C causes each piece of numerical data in the <code>mxArray</code> to be represented as a 16-bit signed integer. In Fortran, use the function <code>mxClassIDFromClassname</code> to derive the <code>classid</code> value from a MATLAB class name. See the Description section for more information.</p> <p><b>ComplexFlag</b> If the data you plan to put into the <code>mxArray</code> has no imaginary components, specify <code>mxREAL</code> in C (0 in Fortran). If the data has some imaginary components, specify <code>mxCOMPLEX</code> in C (1 in Fortran).</p>
<b>Returns</b>	A pointer to the created <code>mxArray</code> , if successful. <code>mxCreateNumericMatrix</code> is unsuccessful if there is not enough free heap space to create the <code>mxArray</code> . If <code>mxCreateNumericMatrix</code> is unsuccessful in a MEX-file, the MEX-file prints an “Out of Memory” message, terminates, and control returns to the MATLAB prompt. If <code>mxCreateNumericMatrix</code>

## mxCreateNumericMatrix (C and Fortran)

---

is unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateNumericMatrix` returns NULL in C (0 in Fortran).

### Description

Call `mxCreateNumericMatrix` to create a 2-D `mxArray` in which all data elements have the numeric data type specified by `classid`. After creating the `mxArray`, `mxCreateNumericMatrix` initializes all its real data elements to 0. If `ComplexFlag` equals `mxCOMPLEX` in C (1 in Fortran), `mxCreateNumericMatrix` also initializes all its imaginary data elements to 0. `mxCreateNumericMatrix` allocates dynamic memory to store the created `mxArray`. When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

The following table shows the C `classid` values and the Fortran data types that are equivalent to MATLAB classes.

<b>MATLAB Class Name</b>	<b>C classid Value</b>	<b>Fortran Type</b>
<code>int8</code>	<code>mxINT8_CLASS</code>	BYTE
<code>uint8</code>	<code>mxUINT8_CLASS</code>	
<code>int16</code>	<code>mxUINT16_CLASS</code>	INTEGER*2
<code>uint16</code>	<code>mxUINT16_CLASS</code>	
<code>int32</code>	<code>mxINT32_CLASS</code>	INTEGER*4
<code>uint32</code>	<code>mxUINT32_CLASS</code>	
<code>int64</code>	<code>mxINT64_CLASS</code>	INTEGER*8
<code>uint64</code>	<code>mxUINT64_CLASS</code>	
<code>single</code>	<code>mxSINGLE_CLASS</code>	REAL*4
<code>double</code>	<code>mxDOUBLE_CLASS</code>	REAL*8

# mxCreateNumericMatrix (C and Fortran)

<b>MATLAB Class Name</b>	<b>C classid Value</b>	<b>Fortran Type</b>
single, with imaginary components	mxSINGLE_CLASS	COMPLEX*8
double, with imaginary components	mxDOUBLE_CLASS	COMPLEX*16

## Fortran Examples

To create a 4-by-3 matrix of REAL\*4 elements having no imaginary components, use

```
C      Create 4x3 mxArray of REAL*4
      mxCreateNumericMatrix(4, 3,
+                          mxClassIDFromClassName('single'), 0)
```

## See Also

`mxClassId`, `mxClassIdFromClassName`, `mxComplexity`,  
`mxCreateNumericArray`

# mxCreateSparse (C and Fortran)

---

<b>Purpose</b>	Create 2-D unpopulated sparse mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateSparse(mwSize m, mwSize n, mwSize nzmax,                         mxComplexity ComplexFlag);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateSparse(m, n, nzmax, ComplexFlag) mwSize m, n, nzmax integer*4 ComplexFlag</pre>
<b>Arguments</b>	<p><b>m</b> The desired number of rows</p> <p><b>n</b> The desired number of columns</p> <p><b>nzmax</b> The number of elements that mxCreateSparse should allocate to hold the pr, ir, and, if ComplexFlag is mxCOMPLEX in C (1 in Fortran), pi arrays. Set the value of nzmax to be greater than or equal to the number of nonzero elements you plan to put into the mxArray, but make sure that nzmax is less than or equal to m*n.</p> <p><b>ComplexFlag</b> If the mxArray you are creating is to contain imaginary data, set ComplexFlag to mxCOMPLEX in C (1 in Fortran). Otherwise, set ComplexFlag to mxREAL in C (0 in Fortran).</p>
<b>Returns</b>	A pointer to the created sparse double mxArray if successful, and NULL in C (0 in Fortran) otherwise. The most likely reason for failure is insufficient free heap space. If that happens, try reducing nzmax, m, or n.
<b>Description</b>	Call mxCreateSparse to create an unpopulated sparse double mxArray. The returned sparse mxArray contains no sparse information and cannot be passed as an argument to any MATLAB sparse functions. To make the returned sparse mxArray useful, you must initialize the pr, ir, jc, and (if it exists) pi arrays.

mxCreateSparse allocates space for

- A `pr` array of length `nzmax`.
- A `pi` array of length `nzmax`, but only if `ComplexFlag` is `mxCOMPLEX` in C (1 in Fortran).
- An `ir` array of length `nzmax`.
- A `jc` array of length `n+1`.

When you finish using the sparse `mxArray`, call `mxDestroyArray` to reclaim all its heap space.

## **C Examples**

See `fulltosparse.c` in the `refbook` subdirectory of the `examples` directory.

## **See Also**

`mxDestroyArray`, `mxSetNzmax`, `mxSetPr`, `mxSetPi`, `mxSetIr`, `mxSetJc`, `mxComplexity`

# mxCreateSparseLogicalMatrix (C)

---

**Purpose** Create unpopulated 2-D, sparse, logical mxArray

**C Syntax**

```
#include "matrix.h"
mxArray *mxCreateSparseLogicalMatrix(mwSize m, mwSize n,
    mwSize nzmax);
```

**Arguments**

m  
The desired number of rows

n  
The desired number of columns

nzmax  
The number of elements that `mxCreateSparseLogicalMatrix` should allocate to hold the data. Set the value of `nzmax` to be greater than or equal to the number of nonzero elements you plan to put into the mxArray, but make sure that `nzmax` is less than or equal to  $m*n$ .

**Returns** A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateSparseLogicalMatrix` returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. `mxCreateSparseLogicalMatrix` is unsuccessful when there is not enough free heap space to create the mxArray.

**Description** Use `mxCreateSparseLogicalMatrix` to create an m-by-n mxArray of logical 1 (true) and logical 0 (false) elements. `mxCreateSparseLogicalMatrix` initializes each element in the array to logical 0.

Call `mxDestroyArray` when you finish using the mxArray. `mxDestroyArray` deallocates the mxArray and its elements.

**See Also** `mxCreateLogicalArray`, `mxCreateLogicalMatrix`, `mxCreateLogicalScalar`, `mxCreateSparse`, `mxIsLogical`



# mxCreateString (C and Fortran)

---

<b>Purpose</b>	Create 1-by-N string mxArray initialized to specified string
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxCreateString(const char *str);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxCreateString(str) character*(*) str</pre>
<b>Arguments</b>	<p>str</p> <p>The string that is to serve as the mxArray's initial data</p>
<b>Returns</b>	A pointer to the created string mxArray if successful, and NULL in C (0 in Fortran) otherwise. The most likely cause of failure is insufficient free heap space.
<b>Description</b>	<p>Use mxCreateString to create a string mxArray initialized to str. Many MATLAB functions (for example, strcmp and upper) require string array inputs.</p> <p>Free the string mxArray when you are finished using it. To free a string mxArray, call mxDestroyArray.</p>
<b>C Examples</b>	<p>See revord.c in the refbook subdirectory of the examples directory.</p> <p>For additional examples, see mxcreatestructarray.c and mxisclass.c in the mx subdirectory of the examples directory.</p>
<b>Fortran Examples</b>	See matdemo1.F in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.
<b>See Also</b>	mxCreateCharMatrixFromStrings, mxCreateCharArray

# mxCreateStructArray (C and Fortran)

---

**Purpose** Create unpopulated N-D structure mxArray

**C Syntax**

```
#include "matrix.h"
mxArray *mxCreateStructArray(mwSize ndim, const mwSize *dims,
    int nfields, const char **fieldnames);
```

**Fortran Syntax**

```
mwPointer function mxCreateStructArray(ndim, dims, nfields,
    fieldnames)
mwSize ndim, dims
integer*4 nfields
character*(*) fieldnames(nfields)
```

**Arguments**

**ndim**  
Number of dimensions. If you set ndim to be less than 2, mxCreateNumericArray creates a two-dimensional mxArray.

**dims**  
The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxArray. In Fortran, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 mxArray. Typically, the dims array should have ndim elements.

**nfields**  
The desired number of fields in each element

**fieldnames**  
The desired list of field names

Each structure field name must begin with a letter and is case sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the namelengthmax function to determine the maximum length of a field name.

# mxCreateStructArray (C and Fortran)

---

**Returns** A pointer to the created structure `mxArray` if successful, and `NULL` in C (0 in Fortran) otherwise. The most likely cause of failure is insufficient heap space to hold the returned `mxArray`.

**Description** Call `mxCreateStructArray` to create an unpopulated structure `mxArray`. Each element of a structure `mxArray` contains the same number of fields (specified in `nfields`). Each field has a name; the list of names is specified in `fieldnames`. A structure `mxArray` in MATLAB is conceptually identical to an array of structs in the C language.

Each field holds one `mxArray` pointer. `mxCreateStructArray` initializes each field to `NULL` in C (0 in Fortran). Call `mxSetField` or `mxSetFieldByNumber` to place a non-`NULL` `mxArray` pointer in a field.

When you finish using the returned structure `mxArray`, call `mxDestroyArray` to reclaim its space.

Any trailing singleton dimensions specified in the `dims` argument are automatically removed from the resulting array. For example, if `ndim` equals 5 and `dims` equals `[4 1 7 1 1]`, the resulting array is given the dimensions 4-by-1-by-7.

**C Examples** See `mxcreatestructarray.c` in the `mx` subdirectory of the `examples` directory.

**See Also** `mxDestroyArray`, `mxAddField`, `mxRemoveField`, `mxSetField`, `mxSetFieldByNumber`

# mxCreateStructMatrix (C and Fortran)

---

**Purpose** Create unpopulated 2-D structure mxArray

**C Syntax**

```
#include "matrix.h"
mxArray *mxCreateStructMatrix(mwSize m, mwSize n, int nfields,
                               const char **fieldnames);
```

**Fortran Syntax**

```
mwPointer function mxCreateStructMatrix(m, n, nfields, fieldnames)
mwSize m, n
integer*4 nfields
character*(*) fieldnames(nfields)
```

**Arguments**

m  
The desired number of rows. This must be a positive integer.

n  
The desired number of columns. This must be a positive integer.

nfields  
The desired number of fields in each element.

fieldnames  
The desired list of field names.

Each structure field name must begin with a letter and is case sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the `nameLengthmax` function to determine the maximum length of a field name.

**Returns** A pointer to the created structure mxArray if successful, and NULL in C (0 in Fortran) otherwise. The most likely cause of failure is insufficient heap space to hold the returned mxArray.

**Description** `mxCreateStructMatrix` and `mxCreateStructArray` are almost identical. The only difference is that `mxCreateStructMatrix` can create only two-dimensional mxArrays, while `mxCreateStructArray` can create mxArrays having two or more dimensions.

# mxCreateStructMatrix (C and Fortran)

---

## **C Examples**

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

## **See Also**

`mxCreateStructArray`

# mxDestroyArray (C and Fortran)

---

**Purpose** Free dynamic memory allocated by mxCreate

**C Syntax**

```
#include "matrix.h"
void mxDestroyArray(mxArray *pm);
```

**Fortran Syntax**

```
subroutine mxDestroyArray(pm)
mwPointer pm
```

**Arguments** pm  
Pointer to the mxArray you want to free

**Description** mxDestroyArray deallocates the memory occupied by the specified mxArray. mxDestroyArray not only deallocates the memory occupied by the mxArray's characteristics fields (such as m and n), but also deallocates all the mxArrays associated data arrays (such as pr, pi, ir, and jc). Do not call mxDestroyArray on an mxArray you are returning on the left-hand side.

**C Examples** See sincall.c in the refbook subdirectory of the examples directory.  
Additional examples:

- mexcallmatlab.c and mexgetarray.c in the mex subdirectory of the examples directory
- mxisclass.c in the mx subdirectory of the examples directory

**See Also** mxCalloc, mxMalloc, mxFree, mexMakeArrayPersistent, mexMakeMemoryPersistent

# mxDuplicateArray (C and Fortran)

---

**Purpose** Make deep copy of array

**C Syntax**

```
#include "matrix.h"
mxArray *mxDuplicateArray(const mxArray *in);
```

**Fortran Syntax**

```
mwPointer function mxDuplicateArray(in)
mwPointer in
```

**Arguments** `in`  
Pointer to the mxArray you want to copy

**Returns** Pointer to a copy of the array.

**Description** `mxDuplicateArray` makes a deep copy of an array, and returns a pointer to the copy. A deep copy refers to a copy in which all levels of data are copied. For example, a deep copy of a cell array copies each cell and the contents of each cell (if any), and so on.

**C Examples** See

- `mexget.c` in the `mex` subdirectory of the examples directory
- `phonebook.c` in the `refbook` subdirectory of the examples directory

For additional examples, see `mxcreatecellmatrix.c`, `mxgetinf.c`, and `mxsetnzmax.c` in the `mx` subdirectory of the examples directory.

# mxFree (C and Fortran)

---

**Purpose** Free dynamic memory allocated by `mxMalloc`, `mxRealloc`

**C Syntax**

```
#include "matrix.h"
void mxFree(void *ptr);
```

**Fortran Syntax**

```
subroutine mxFree(ptr)
mwPointer ptr
```

**Arguments** `ptr`  
Pointer to the beginning of any memory parcel allocated by `mxMalloc`, `mxRealloc`.

**Description** `mxFree` deallocates heap space using the MATLAB memory management facility. This ensures correct memory management in error and abort (**Ctrl+C**) conditions.

To deallocate heap space, MATLAB applications in C should always call `mxFree` rather than the ANSI C `free` function.

The MATLAB memory management facility maintains a list of all memory allocated by `mxMalloc`, `mxRealloc`, and the `mxCreate*` calls. The MATLAB memory management facility automatically deallocates all of a MEX-file's managed parcels when the MEX-file completes and control returns to the MATLAB prompt.

When `mxFree` appears in a stand-alone MATLAB application, `mxFree` simply deallocates the contiguous heap space that begins at address `ptr`. In a MEX-file, `mxFree` also removes the memory parcel from the MATLAB memory management facility's list of memory parcels.

In a MEX-file, your use of `mxFree` depends on whether the specified memory parcel is persistent or nonpersistent. By default, memory parcels created by `mxMalloc`, `mxRealloc` are nonpersistent. The MATLAB memory management facility automatically frees all nonpersistent memory whenever a MEX-file completes. Thus, even if you do not call `mxFree`, MATLAB takes care of freeing the memory for you. Nevertheless, it is good programming



practice to deallocate memory as soon as you are through using it. Doing so generally makes the entire system run more efficiently.

If an application calls `mexMakeMemoryPersistent`, the specified memory parcel becomes persistent. When a MEX-file completes, the MATLAB memory management facility does not free persistent memory parcels. Therefore, the only way to free a persistent memory parcel is to call `mxFree`. Typically, MEX-files call `mexAtExit` to register a cleanup handler. The cleanup handler calls `mxFree`.

### C Examples

See `mxcalcsinglesubscript.c` in the `mx` subdirectory of the examples directory.

Additional examples:

- `phonebook.c` in the `refbook` subdirectory of the examples directory
- `explore.c` and `mexatexit.c` in the `mex` subdirectory of the examples directory
- `mxcreatecharmatrixfromstr.c`, `mxisfinite.c`, `mxmalloc.c`, and `mxsetdimensions.c` in the `mx` subdirectory of the examples directory

### See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxMalloc`, `mxRealloc`

# mxGetCell (C and Fortran)

---

<b>Purpose</b>	Contents of mxArray cell
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxGetCell(const mxArray *pm, mwIndex index);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetCell(pm, index) mwPointer pm mwIndex index</pre>
<b>Arguments</b>	<p>pm Pointer to a cell mxArray</p> <p>index The number of elements in the cell mxArray between the first element and the desired one. See mxCalcSingleSubscript for details on calculating an index in a multidimensional cell array.</p>
<b>Returns</b>	<p>A pointer to the <i>i</i>th cell mxArray if successful, and NULL in C (0 in Fortran) otherwise. Causes of failure include</p> <ul style="list-style-type: none"><li>• Specifying the index of a cell array element that has not been populated.</li><li>• Specifying a pm that does not point to a cell mxArray.</li><li>• Specifying an index greater than the number of elements in the cell.</li><li>• Insufficient free heap space to hold the returned cell mxArray.</li></ul>
<b>Description</b>	<p>Call mxGetCell to get a pointer to the mxArray held in the indexed element of the cell mxArray.</p> <hr/> <p><b>Note</b> Inputs to a MEX-file are constant read-only mxArrays and should not be modified. Using mxSetCell* or mxSetField* to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.</p> <hr/>

### **C Examples**

See `explore.c` in the `mex` subdirectory of the `examples` directory.

### **See Also**

`mxCreateCellArray`, `mxIsCell`, `mxSetCell`

## mxGetChars (C)

---

<b>Purpose</b>	Pointer to character array data
<b>C Syntax</b>	<pre>#include "matrix.h" mxChar *mxGetChars(const mxArray *array_ptr);</pre>
<b>Arguments</b>	array_ptr Pointer to an mxArray
<b>Returns</b>	The address of the first character in the mxArray. Returns NULL if the specified array is not a character array.
<b>Description</b>	Call mxGetChars to determine the address of the first character in the mxArray that array_ptr points to. Once you have the starting address, you can access any other element in the mxArray.
<b>See Also</b>	mxGetString

<b>Purpose</b>	Class of mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mxClassID mxGetClassID(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxGetClassID(pm) mwPointer pm</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p>
<b>Returns</b>	<p>A numeric identifier of the class (category) of the mxArray that pm points to. The C-language class identifiers are</p> <p>mxUNKNOWN_CLASS     The class cannot be determined. You cannot specify this category for an mxArray; however, mxGetClassID can return this value if it cannot identify the class.</p> <p>mxCELL_CLASS     Identifies a cell mxArray.</p> <p>mxSTRUCT_CLASS     Identifies a structure mxArray.</p> <p>mxCHAR_CLASS     Identifies a string mxArray, an mxArray whose data is represented as mxCHAR's.</p> <p>mxLOGICAL_CLASS     Identifies a logical mxArray, an mxArray that stores the logical values 1 and 0, representing the states true and false respectively.</p> <p>mxDOUBLE_CLASS     Identifies a numeric mxArray whose data is stored as double-precision, floating-point numbers.</p>

## mxGetClassID (C and Fortran)

---

<code>mxSINGLE_CLASS</code>	Identifies a numeric mxArray whose data is stored as single-precision, floating-point numbers.
<code>mxINT8_CLASS</code>	Identifies a numeric mxArray whose data is stored as signed 8-bit integers.
<code>mxUINT8_CLASS</code>	Identifies a numeric mxArray whose data is stored as unsigned 8-bit integers.
<code>mxINT16_CLASS</code>	Identifies a numeric mxArray whose data is stored as signed 16-bit integers.
<code>mxUINT16_CLASS</code>	Identifies a numeric mxArray whose data is stored as unsigned 16-bit integers.
<code>mxINT32_CLASS</code>	Identifies a numeric mxArray whose data is stored as signed 32-bit integers.
<code>mxUINT32_CLASS</code>	Identifies a numeric mxArray whose data is stored as unsigned 32-bit integers.
<code>mxINT64_CLASS</code>	Identifies a numeric mxArray whose data is stored as signed 64-bit integers.
<code>mxUINT64_CLASS</code>	Identifies a numeric mxArray whose data is stored as unsigned 64-bit integers.
<code>mxFUNCTION_CLASS</code>	Identifies a function handle mxArray.

### Description

Use `mxGetClassId` to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For

example, if `pm` points to a logical `mxArray`, then `mxGetClassID` returns `mxLOGICAL_CLASS` (in C).

`mxGetClassID` is similar to `mxGetClassName`, except that the former returns the class as an integer identifier and the latter returns the class as a string.

### **C Examples**

See

- `phonebook.c` in the `refbook` subdirectory of the `examples` directory
- `explore.c` in the `mex` subdirectory of the `examples` directory

### **See Also**

`mxGetClassName`

# mxGetClassName (C and Fortran)

---

<b>Purpose</b>	Class of mxArray as string
<b>C Syntax</b>	<pre>#include "matrix.h" const char *mxGetClassName(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>character*(*) function mxGetClassName(pm) mwPointer pm</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p>
<b>Returns</b>	The class (as a string) of the mxArray pointed to by pm.
<b>Description</b>	<p>Call mxGetClassName to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For example, if pm points to a logical mxArray, mxGetClassName returns logical.</p> <p>mxGetClassID is similar to mxGetClassName, except that the former returns the class as an integer identifier and the latter returns the class as a string.</p>
<b>C Examples</b>	<p>See mexfunction.c in the mex subdirectory of the examples directory. For an additional example, see mxisclass.c in the mx subdirectory of the examples directory.</p>
<b>See Also</b>	mxGetClassID



<b>Purpose</b>	Pointer to data
<b>C Syntax</b>	<pre>#include "matrix.h" void *mxGetData(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetData(pm) mwPointer pm</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p>
<b>Returns</b>	The address of the first element of the real data. Returns NULL in C (0 in Fortran) if there is no real data.
<b>Description</b>	<p>Similar to mxGetPr, except that in C, mxGetData returns a void *.</p> <p>To copy values from the returned pointer to Fortran, use one of the mxCopyPtrTo* functions in the following manner:</p> <pre>    C        Get the data in mxArray, pm             mxCopyPtrToReal8(mxGetData(pm), data,             +                    mxGetNumberOfElements(pm))</pre>
<b>C Examples</b>	<p>See phonebook.c in the refbook subdirectory of the examples directory.</p> <p>For additional examples, see mxcreatecharmatrixfromstr.c and mxisfinite.c in the mx subdirectory of the examples directory.</p>
<b>See Also</b>	mxGetImagData, mxGetPr

# mxGetDimensions (C and Fortran)

---

**Purpose** Pointer to dimensions array

**C Syntax**

```
#include "matrix.h"
const mwSize *mxGetDimensions(const mxArray *pm);
```

**Fortran Syntax**

```
mwPointer function mxGetDimensions(pm)
mwPointer pm
```

**Arguments** pm  
Pointer to an mxArray.

**Returns** The address of the first element in the dimensions array. Each integer in the dimensions array represents the number of elements in a particular dimension. The array is not NULL terminated.

**Description** Use mxGetDimensions to determine how many elements are in each dimension of the mxArray that pm points to. Call mxGetNumberOfDimensions to get the number of dimensions in the mxArray.

To copy the values to Fortran, use mxCopyPtrToInteger4 in the following manner:

```
C      Get dimensions of mxArray, pm
      mxCopyPtrToInteger4(mxGetDimensions(pm), dims,
+                          mxGetNumberOfDimensions(pm))
```

**C Examples** See mxcalcsinglesubscript.c in the mx subdirectory of the examples directory.

Additional examples:

- findnz.c and phonebook.c in the refbook subdirectory of the examples directory
- explore.c in the mex subdirectory of the examples directory

## mxGetDimensions (C and Fortran)

---

- `mxgeteps.c` and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory

### See Also

`mxGetNumberOfDimensions`

# mxGetElementSize (C and Fortran)

---

**Purpose** Number of bytes required to store each data element

**C Syntax**

```
#include "matrix.h"
size_t mxGetElementSize(const mxArray *pm);
```

**Fortran Syntax**

```
mwSize function mxGetElementSize(pm)
mwPointer pm
```

**Arguments** pm  
Pointer to an mxArray

**Returns** The number of bytes required to store one element of the specified mxArray, if successful. Returns 0 on failure. The primary reason for failure is that pm points to an mxArray having an unrecognized class. If pm points to a cell mxArray or a structure mxArray, mxGetElementSize returns the size of a pointer (not the size of all the elements in each cell or structure field).

**Description** Call mxGetElementSize to determine the number of bytes in each data element of the mxArray. For example, if the MATLAB class of an mxArray is int16, the mxArray stores each data element as a 16-bit (2-byte) signed integer. Thus, mxGetElementSize returns 2.

mxGetElementSize is particularly helpful when using a non-MATLAB routine to manipulate data elements. For example, the C function memcpy requires (for its third argument) the size of the elements you intend to copy.

**C Examples** See doubleelement.c and phonebook.c in the refbook subdirectory of the examples directory.

**See Also** mxGetM, mxGetN

<b>Purpose</b>	Value of eps
<b>C Syntax</b>	<pre>#include "matrix.h" double mxGetEps(void);</pre>
<b>Fortran Syntax</b>	<pre>real*8 function mxGetEps</pre>
<b>Returns</b>	The value of the MATLAB eps variable
<b>Description</b>	Call mxGetEps to return the value of the MATLAB eps variable. This variable holds the distance from 1.0 to the next largest floating-point number. As such, it is a measure of floating-point accuracy. The MATLAB PINV and RANK functions use eps as a default tolerance.
<b>C Examples</b>	See mxgeteps.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxGetInf, mxGetNan

# mxGetField (C and Fortran)

---

## Purpose

Field value, given field name and index into structure array

## C Syntax

```
#include "matrix.h"
mxArray *mxGetField(const mxArray *pm, mwIndex index,
                    const char *fieldname);
```

## Fortran Syntax

```
mwPointer function mxGetField(pm, index, fieldname)
mwPointer pm
mwIndex index
character*(*) fieldname
```

## Arguments

pm

Pointer to a structure mxArray

index

The desired element. In C, the first element of an mxArray has an index of 0, the second element has an index of 1, and the last element has an index of N-1, where N is the total number of elements in the structure mxArray. In Fortran, the first element of an mxArray has an index of 1, the second element has an index of 2, and the last element has an index of N, where N is the total number of elements in the structure mxArray.

fieldname

The name of the field whose value you want to extract.

## Returns

A pointer to the mxArray in the specified field at the specified fieldname, on success. Returns NULL in C (0 in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying an array pointer pm that does not point to a structure mxArray. To determine whether pm points to a structure mxArray, call mxIsStruct.
- Specifying an index to an element outside the bounds of the mxArray. For example, given a structure mxArray that contains 10 elements, you cannot specify an index greater than 9 in C (10 in Fortran).

- Specifying a nonexistent fieldname. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to get existing field names.
- Insufficient heap space to hold the returned `mxArray`.

## Description

Call `mxGetField` to get the value held in the specified element of the specified field. In pseudo-C terminology, `mxGetField` returns the value at

```
pm[index].fieldname
```

`mxGetFieldByNumber` is similar to `mxGetField`. Both functions return the same value. The only difference is in the way you specify the field. `mxGetFieldByNumber` takes a field number as its third argument, and `mxGetField` takes a field name as its third argument.

---

**Note** Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

---

In C, calling

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");  
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is 0 if you have a 1-by-1 structure.

In Fortran, calling

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling

## mxGetField (C and Fortran)

---

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where index is 1 if you have a 1-by-1 structure.

### **See Also**

mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber,  
mxGetNumberOfFields, mxIsStruct, mxSetField, mxSetFieldByNumber



# mxGetFieldByNumber (C and Fortran)

---

<b>Purpose</b>	Field value, given field number and index into structure array
<b>C Syntax</b>	<pre>#include "matrix.h" mxArray *mxGetFieldByNumber(const mxArray *pm, mwIndex index,                              int fieldnumber);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetFieldByNumber(pm, index, fieldnumber) mwPointer pm mwIndex index integer*4 fieldnumber</pre>
<b>Arguments</b>	<p><b>pm</b> Pointer to a structure mxArray</p> <p><b>index</b> The desired element. In C, the first element of an mxArray has an index of 0, the second element has an index of 1, and the last element has an index of N-1, where N is the total number of elements in the structure mxArray. In Fortran, the first element of an mxArray has an index of 1, the second element has an index of 2, and the last element has an index of N, where N is the total number of elements in the structure mxArray. See <code>mxCalcSingleSubscript</code> for more details on calculating an index.</p> <p><b>fieldnumber</b> The position of the field whose value you want to extract. In C, the first field within each element has a field number of 0, the second field has a field number of 1, and so on. The last field has a field number of N-1, where N is the number of fields. In Fortran, the first field within each element has a field number of 1, the second field has a field number of 2, and so on. The last field has a field number of N, where N is the number of fields.</p>
<b>Returns</b>	A pointer to the mxArray in the specified field for the desired element, on success. Returns NULL in C (0 in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

## mxGetFieldByNumber (C and Fortran)

---

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.
- Specifying an index to an element outside the bounds of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an index greater than 9 in C (10 in Fortran).
- Specifying a nonexistent field number. Call `mxGetFieldNumber` to determine the field number that corresponds to a given field name.

### Description

Call `mxGetFieldByNumber` to get the value held in the specified `fieldnumber` at the indexed element.

---

**Note** Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

---

In C, calling

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");  
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is 0 if you have a 1-by-1 structure.

In Fortran, calling

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')  
mxGetFieldByNumber(pm, index, fieldnum)
```

# mxGetFieldByNumber (C and Fortran)

---

where `index` is 1 if you have a 1-by-1 structure.

## **C Examples**

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `mxiclass.c` in the `mx` subdirectory of the `examples` directory
- `explore.c` in the `mex` subdirectory of the `examples` directory

## **See Also**

`mxGetField`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`,  
`mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

# mxGetFieldNameByNumber (C and Fortran)

---

<b>Purpose</b>	Field name, given field number in structure array
<b>C Syntax</b>	<pre>#include "matrix.h" const char *mxGetFieldNameByNumber(const mxArray *pm,     int fieldnumber);</pre>
<b>Fortran Syntax</b>	<pre>character*(*) function mxGetFieldNameByNumber(pm, fieldnumber) mwPointer pm integer*4 fieldnumber</pre>
<b>Arguments</b>	<p><code>pm</code> Pointer to a structure mxArray</p> <p><code>fieldnumber</code> The position of the desired field. For instance, in C, to get the name of the first field, set <code>fieldnumber</code> to 0; to get the name of the second field, set <code>fieldnumber</code> to 1; and so on. In Fortran, to get the name of the first field, set <code>fieldnumber</code> to 1; to get the name of the second field, set <code>fieldnumber</code> to 2; and so on.</p>
<b>Returns</b>	<p>A pointer to the <code>n</code>th field name, on success. Returns NULL in C (0 in Fortran) on failure. Common causes of failure include</p> <ul style="list-style-type: none"><li>• Specifying an array pointer <code>pm</code> that does not point to a structure mxArray. Call <code>mxIsStruct</code> to determine whether <code>pm</code> points to a structure mxArray.</li><li>• Specifying a value of <code>fieldnumber</code> outside the bounds of the number of fields in the structure mxArray. In C, <code>fieldnumber</code> 0 represents the first field, and <code>fieldnumber</code> <code>N</code>-1 represents the last field, where <code>N</code> is the number of fields in the structure mxArray. In Fortran, <code>fieldnumber</code> 1 represents the first field, and <code>fieldnumber</code> <code>N</code> represents the last field.</li></ul>
<b>Description</b>	Call <code>mxGetFieldNameByNumber</code> to get the name of a field in the given structure mxArray. A typical use of <code>mxGetFieldNameByNumber</code> is to call

# mxGetFieldNameByNumber (C and Fortran)

---

it inside a loop in order to get the names of all the fields in a given mxArray.

Consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field number 0 represents the field name; field number 1 represents field billing; field number 2 represents field test. A field number other than 0, 1, or 2 causes mxGetFieldNameByNumber to return NULL.

In Fortran, the field number 1 represents the field name; field number 2 represents field billing; field number 3 represents field test. A field number other than 1, 2, or 3 causes mxGetFieldNameByNumber to return 0.

## C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `mxisclass.c` in the `mx` subdirectory of the `examples` directory
- `explore.c` in the `mex` subdirectory of the `examples` directory

## See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNumber`,  
`mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

# mxGetFieldName (C and Fortran)

---

**Purpose** Field number, given field name in structure array

**C Syntax**

```
#include "matrix.h"
int mxGetFieldName(const mxArray *pm,
    const char *fieldname);
```

**Fortran Syntax**

```
integer*4 function mxGetFieldName(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

**Arguments**

pm  
    Pointer to a structure mxArray.

fieldname  
    The name of a field in the structure mxArray.

**Returns** The field number of the specified fieldname, on success. In C, the first field has a field number of 0, the second field has a field number of 1, and so on. In Fortran, the first field has a field number of 1, the second field has a field number of 2, and so on. Returns -1 in C (0 in Fortran) on failure. Common causes of failure include

- Specifying an array pointer pm that does not point to a structure mxArray. Call mxIsStruct to determine whether pm points to a structure mxArray.
- Specifying the fieldname of a nonexistent field.

**Description** If you know the name of a field but do not know its field number, call mxGetFieldName. Conversely, if you know the field number but do not know its field name, call mxGetFieldNameByNumber.

For example, consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

# mxGetFieldName (C and Fortran)

---

In C, the field name has a field number of 0; the field `billing` has a field number of 1; and the field `test` has a field number of 2. If you call `mxGetFieldName` and specify a field name of anything other than `name`, `billing`, or `test`, `mxGetFieldName` returns -1.

Calling

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling

```
field_num = mxGetFieldName(pa, "field_name");  
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is 0 if you have a 1-by-1 structure.

In Fortran, the field name has a field number of 1; the field `billing` has a field number of 2; and the field `test` has a field number of 3. If you call `mxGetFieldName` and specify a field name of anything other than `'name'`, `'billing'`, or `'test'`, `mxGetFieldName` returns 0.

Calling

```
mxGetField(pm, index, 'fieldname');
```

is equivalent to calling

```
fieldnum = mxGetFieldName(pm, 'fieldname');  
mxGetFieldByNumber(pm, index, fieldnum);
```

where `index` is 1 if you have a 1-by-1 structure.

## C Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the examples directory.

## See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`,  
`mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

# mxGetImagData (C and Fortran)

---

<b>Purpose</b>	Pointer to imaginary data of mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" void *mxGetImagData(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetImagData(pm) mwPointer pm</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p>
<b>Returns</b>	The address of the first element of the imaginary data, on success. Returns NULL in C (0 in Fortran) if there is no imaginary data or if there is an error.
<b>Description</b>	This function is similar to mxGetPi, except that in C it returns a void *.
<b>C Examples</b>	See mxisfinite.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxGetData, mxGetPi



<b>Purpose</b>	Value of infinity
<b>C Syntax</b>	<pre>#include "matrix.h" double mxGetInf(void);</pre>
<b>Fortran Syntax</b>	<pre>real*8 function mxGetInf</pre>
<b>Returns</b>	The value of infinity on your system.
<b>Description</b>	<p>Call <code>mxGetInf</code> to return the value of the MATLAB internal <code>inf</code> variable. <code>inf</code> is a permanent variable representing IEEE arithmetic positive infinity. The value of <code>inf</code> is built into the system; you cannot modify it.</p> <p>Operations that return infinity include</p> <ul style="list-style-type: none"><li>• Division by 0. For example, <code>5/0</code> returns infinity.</li><li>• Operations resulting in overflow. For example, <code>exp(10000)</code> returns infinity because the result is too large to be represented on your machine.</li></ul>
<b>C Examples</b>	See <code>mxgetinf.c</code> in the <code>mx</code> subdirectory of the <code>examples</code> directory.
<b>See Also</b>	<code>mxGetEps</code> , <code>mxGetNaN</code>

# mxGetIr (C and Fortran)

---

**Purpose**            `ir` array of sparse matrix

**C Syntax**            `#include "matrix.h"`  
`mwIndex *mxGetIr(const mxArray *pm);`

**Fortran Syntax**        `mwPointer function mxGetIr(pm)`  
`mwPointer pm`

**Arguments**        `pm`  
                        Pointer to a sparse `mxArray`

**Returns**            A pointer to the first element in the `ir` array, if successful, and NULL in C (0 in Fortran) otherwise. Possible causes of failure include

- Specifying a full (nonsparse) `mxArray`.
- Specifying a value for `pm` that is NULL in C (0 in Fortran). This usually means that an earlier call to `mxCreateSparse` failed.

**Description**        Use `mxGetIr` to obtain the starting address of the `ir` array. The `ir` array is an array of integers; the length of the `ir` array is typically `nzmax` values. For example, if `nzmax` equals 100, the `ir` array should contain 100 integers.

Each value in an `ir` array indicates a row (offset by 1) at which a nonzero element can be found. (The `jc` array is an index that indirectly specifies a column where nonzero elements can be found.)

For details on the `ir` and `jc` arrays, see `mxSetIr` and `mxSetJc`.

**C Examples**            See `fulltosparse.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `explore.c` in the `mex` subdirectory of the `examples` directory

- `mxsetdimensions.c` and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory

### See Also

`mxGetJc`, `mxGetNzmax`, `mxSetIr`, `mxSetJc`, `mxSetNzmax`

# mxGetJc (C and Fortran)

---

<b>Purpose</b>	jc array of sparse matrix
<b>C Syntax</b>	<pre>#include "matrix.h" mwIndex *mxGetJc(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetJc(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to a sparse mxArray
<b>Returns</b>	A pointer to the first element in the jc array, if successful, and NULL in C (0 in Fortran) otherwise. Possible causes of failure include <ul style="list-style-type: none"><li>• Specifying a full (nonsparse) mxArray.</li><li>• Specifying a value for pm that is NULL in C (0 in Fortran). This usually means that an earlier call to mxCreateSparse failed.</li></ul>
<b>Description</b>	Use mxGetJc to obtain the starting address of the jc array. The jc array is an integer array having n+1 elements, where n is the number of columns in the sparse mxArray. The values in the jc array indirectly indicate columns containing nonzero elements. For a detailed explanation of the jc array, see mxSetJc.
<b>C Examples</b>	See fulltosparse.c in the refbook subdirectory of the examples directory. Additional examples: <ul style="list-style-type: none"><li>• explore.c in the mex subdirectory of the examples directory</li><li>• mxgetnzmax.c, mxsetdimensions.c, and mxsetnzmax.c in the mx subdirectory of the examples directory</li></ul>
<b>See Also</b>	mxGetIr, mxGetNzmax, mxSetIr, mxSetJc, mxSetNzmax

<b>Purpose</b>	Pointer to logical array data
<b>C Syntax</b>	<pre>#include "matrix.h" mxLogical *mxGetLogicals(const mxArray *array_ptr);</pre>
<b>Arguments</b>	<pre>array_ptr     Pointer to an mxArray</pre>
<b>Returns</b>	The address of the first logical element in the mxArray. The result is unspecified if the mxArray is not a logical array.
<b>Description</b>	Call <code>mxGetLogicals</code> to determine the address of the first logical element in the mxArray that <code>array_ptr</code> points to. Once you have the starting address, you can access any other element in the mxArray.
<b>See Also</b>	<code>mxCreateLogicalArray</code> , <code>mxCreateLogicalMatrix</code> , <code>mxCreateLogicalScalar</code> , <code>mxIsLogical</code> , <code>mxIsLogicalScalar</code> , <code>mxIsLogicalScalarTrue</code>

# mxGetM (C and Fortran)

---

<b>Purpose</b>	Number of rows in mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" size_t mxGetM(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwSize function mxGetM(pm) mwPointer pm</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p>
<b>Returns</b>	The number of rows in the mxArray to which pm points.
<b>Description</b>	mxGetM returns the number of rows in the specified array. The term <i>rows</i> always means the first dimension of the array, no matter how many dimensions the array has. For example, if pm points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, mxGetM returns 8.
<b>C Examples</b>	<p>See <code>convec.c</code> in the <code>refbook</code> subdirectory of the <code>examples</code> directory.</p> <p>Additional examples:</p> <ul style="list-style-type: none"><li>• <code>fulltospars.c</code>, <code>revord.c</code>, <code>timestwo.c</code>, and <code>xtimesy.c</code> in the <code>refbook</code> subdirectory of the <code>examples</code> directory</li><li>• <code>explore.c</code>, <code>mexget.c</code>, <code>mexlock.c</code>, <code>mexsettrapflag.c</code> and <code>yprime.c</code> in the <code>mex</code> subdirectory of the <code>examples</code> directory</li><li>• <code>mxmalloc.c</code>, <code>mxsetdimensions.c</code>, <code>mxgetnzmax.c</code>, and <code>mxsetnzmax.c</code> in the <code>mx</code> subdirectory of the <code>examples</code> directory</li></ul>
<b>Fortran Examples</b>	See <code>matdemo2.F</code> in the <code>eng_mat</code> subdirectory of the <code>examples</code> directory for a sample program that illustrates how to use this routine in a Fortran program.
<b>See Also</b>	<code>mxGetN</code> , <code>mxSetM</code> , <code>mxSetN</code>

**Purpose** Number of columns in mxArray

**C Syntax**

```
#include "matrix.h"
size_t mxGetN(const mxArray *pm);
```

**Fortran Syntax**

```
mwSize function mxGetN(pm)
mwPointer pm
```

**Arguments** pm  
Pointer to an mxArray

**Returns** The number of columns in the mxArray.

**Description** Call mxGetN to determine the number of columns in the specified mxArray.

If pm is an N-dimensional mxArray, mxGetN is the product of dimensions 2 through N. For example, if pm points to a four-dimensional mxArray having dimensions 13-by-5-by-4-by-6, mxGetN returns the value 120 ( $5 \times 4 \times 6$ ). If the specified mxArray has more than two dimensions and you need to know exactly how many elements are in each dimension, call mxGetDimensions.

If pm points to a sparse mxArray, mxGetN still returns the number of columns, not the number of occupied columns.

**C Examples** See convec.c in the refbook subdirectory of the examples directory.

Additional examples:

- fulltosparse.c, revord.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory
- explore.c, mexget.c, mexlock.c, mexsettrapflag.c and yprime.c in the mex subdirectory of the examples directory
- mxmalloc.c, mxsetdimensions.c, mxgetnzmax.c, and mxsetnzmax.c in the mx subdirectory of the examples directory

## mxGetN (C and Fortran)

---

### **Fortran Examples**

See `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

### **See Also**

`mxGetM`, `mxGetDimensions`, `mxSetM`, `mxSetN`



**Purpose** Value of NaN (Not-a-Number)

**C Syntax**

```
#include "matrix.h"
double mxGetNaN(void);
```

**Fortran Syntax**

```
real*8 function mxGetNaN
```

**Returns** The value of NaN (Not-a-Number) on your system

**Description** Call `mxGetNaN` to return the value of NaN for your system. NaN is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return NaN as a result, for example,

- `0.0/0.0`
- `Inf-Inf`

The value of Not-a-Number is built in to the system. You cannot modify it.

**C Examples** See `mxgetinf.c` in the `mx` subdirectory of the examples directory.

**See Also** `mxGetEps`, `mxGetInf`

# mxGetNumberOfDimensions (C and Fortran)

---

<b>Purpose</b>	Number of dimensions in mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" mwSize mxGetNumberOfDimensions(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwSize function mxGetNumberOfDimensions(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	The number of dimensions in the specified mxArray. The returned value is always 2 or greater.
<b>Description</b>	Use mxGetNumberOfDimensions to determine how many dimensions are in the specified array. To determine how many elements are in each dimension, call mxGetDimensions.
<b>C Examples</b>	See explore.c in the mex subdirectory of the examples directory. Additional examples: <ul style="list-style-type: none"><li>• findnz.c, fulltosparse.c, and phonebook.c in the refbook subdirectory of the examples directory</li><li>• mxcalcsinglesubscript.c, mxgeteps.c, and mxisfinite.c in the mx subdirectory of the examples directory.</li></ul>
<b>See Also</b>	mxSetM, mxSetN, mxGetDimensions

# mxGetNumberOfElements (C and Fortran)

---

<b>Purpose</b>	Number of elements in mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" size_t mxGetNumberOfElements(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwSize function mxGetNumberOfElements(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Number of elements in the specified mxArray
<b>Description</b>	mxGetNumberOfElements tells you how many elements an array has. For example, if the dimensions of an array are 3-by-5-by-10, mxGetNumberOfElements returns the number 150.
<b>C Examples</b>	See findnz.c and phonebook.c in the refbook subdirectory of the examples directory. Additional examples: <ul style="list-style-type: none"><li>• explore.c in the mex subdirectory of the examples directory</li><li>• mxcalcsinglesubscript.c, mxgeteps.c, mxgetinf.c, mxisfinite.c, and mxsetdimensions.c in the mx subdirectory of the examples directory</li></ul>
<b>See Also</b>	mxGetDimensions, mxGetM, mxGetN, mxGetClassID, mxGetClassName

# mxGetNumberOfFields (C and Fortran)

---

<b>Purpose</b>	Number of fields in structure mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" int mxGetNumberOfFields(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxGetNumberOfFields(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to a structure mxArray
<b>Returns</b>	The number of fields, on success. Returns 0 on failure. The most common cause of failure is that pm is not a structure mxArray. Call mxIsStruct to determine whether pm is a structure.
<b>Description</b>	Call mxGetNumberOfFields to determine how many fields are in the specified structure mxArray.  Once you know the number of fields in a structure, you can loop through every field in order to set or to get field values.
<b>C Examples</b>	See phonebook.c in the refbook subdirectory of the examples directory. Additional examples: <ul style="list-style-type: none"><li>• mxisclass.c in the mx subdirectory of the examples directory</li><li>• explore.c in the mex subdirectory of the examples directory.</li></ul>
<b>See Also</b>	mxGetField, mxIsStruct, mxSetField

<b>Purpose</b>	Number of elements in <i>ir</i> , <i>pr</i> , and <i>pi</i> arrays
<b>C Syntax</b>	<pre>#include "matrix.h" mwSize mxGetNzmax(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwSize function mxGetNzmax(pm) mwPointer pm</pre>
<b>Arguments</b>	<p><i>pm</i></p> <p>Pointer to a sparse mxArray</p>
<b>Returns</b>	The number of elements allocated to hold nonzero entries in the specified sparse mxArray, on success. Returns an indeterminate value on error. The most likely cause of failure is that <i>pm</i> points to a full (nonsparse) mxArray.
<b>Description</b>	<p>Use <code>mxGetNzmax</code> to get the value of the <code>nzmax</code> field. The <code>nzmax</code> field holds an integer value that signifies the number of elements in the <i>ir</i>, <i>pr</i>, and, if it exists, the <i>pi</i> arrays. The value of <code>nzmax</code> is always greater than or equal to the number of nonzero elements in a sparse mxArray. In addition, the value of <code>nzmax</code> is always less than or equal to the number of rows times the number of columns.</p> <p>As you adjust the number of nonzero elements in a sparse mxArray, MATLAB often adjusts the value of the <code>nzmax</code> field. MATLAB adjusts <code>nzmax</code> in order to reduce the number of costly reallocations and in order to optimize its use of heap space.</p>
<b>C Examples</b>	See <code>mxgetnzmax.c</code> and <code>mxsetnzmax.c</code> in the <code>mx</code> subdirectory of the <code>examples</code> directory.
<b>See Also</b>	<code>mxSetNzmax</code>

# mxGetPi (C and Fortran)

---

<b>Purpose</b>	Imaginary data elements in mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" double *mxGetPi(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetPi(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	The imaginary data elements of the specified mxArray, on success. Returns NULL in C (0 in Fortran) if there is no imaginary data or if there is an error.
<b>Description</b>	<p>The pi field points to an array containing the imaginary data of the mxArray. Call mxGetPi to get the contents of the pi field, that is, to get the starting address of this imaginary data.</p> <p>The best way to determine whether an mxArray is purely real is to call mxIsComplex.</p> <p>The imaginary parts of all input matrices to a MATLAB function are allocated if any of the input matrices are complex.</p>
<b>C Examples</b>	<p>See convec.c, findnz.c, and fulltosparse.c in the refbook subdirectory of the examples directory.</p> <p>Additional examples:</p> <ul style="list-style-type: none"><li>• explore.c and mexcallmatlab.c in the mex subdirectory of the examples directory</li><li>• mxcalcsinglesubscript.c, mxgetinf.c, mxisfinite.c, and mxsetnzmax.c in the mx subdirectory of the examples directory</li></ul>
<b>See Also</b>	mxGetPr, mxSetPi, mxSetPr

<b>Purpose</b>	Real data elements in mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" double *mxGetPr(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxGetPr(pm) mwPointer pm</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p>
<b>Returns</b>	The address of the first element of the real data. Returns NULL in C (0 in Fortran) if there is no real data.
<b>Description</b>	Call mxGetPr to determine the starting address of the real data in the mxArray that pm points to. Once you have the starting address, you can access any other element in the mxArray.
<b>C Examples</b>	See convec.c, doubleelement.c, findnz.c, fulltosparse.c, sincall.c, timestwo.c, timestwoalt.c, and xtimesy.c in the refbook subdirectory of the examples directory.
<b>See Also</b>	mxGetPi, mxSetPi, mxSetPr

# mxGetScalar (C and Fortran)

---

**Purpose** Real component of first data element in mxArray

**C Syntax**

```
#include "matrix.h"
double mxGetScalar(const mxArray *pm);
```

**Fortran Syntax**

```
real*8 function mxGetScalar(pm)
mwPointer pm
```

**Arguments** pm  
Pointer to an mxArray other than a cell mxArray or a structure mxArray

**Returns** The value of the first real (nonimaginary) element of the mxArray. Notice that in C, mxGetScalar returns a double. Therefore, if real elements in the mxArray are stored as something other than doubles, mxGetScalar automatically converts the scalar value into a double. To preserve the original data representation of the scalar, you must cast the return value to the desired data type.

If pm points to a structure mxArray or a cell mxArray, mxGetScalar returns 0.0.

If pm points to a sparse mxArray, mxGetScalar returns the value of the first nonzero real element in the mxArray.

If pm points to an empty mxArray, mxGetScalar returns an indeterminate value.

**Description** Call mxGetScalar to get the value of the first real (nonimaginary) element of the mxArray.

In most cases, you call mxGetScalar when pm points to an mxArray containing only one element (a scalar). However, pm can point to an mxArray containing many elements. If pm points to an mxArray containing multiple elements, mxGetScalar returns the value of the first real element. If pm points to a two-dimensional mxArray, mxGetScalar returns the value of the (1, 1) element; if pm points to



a three-dimensional mxArray, mxGetScalar returns the value of the (1,1,1) element; and so on.

### **C Examples**

See `timestwoalt.c` and `xtimesy.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory
- `mexget.c`, `mexlock.c` and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory

### **See Also**

`mxGetM`, `mxGetN`

# mxGetString (C and Fortran)

---

<b>Purpose</b>	Copy string mxArray to C-style string
<b>C Syntax</b>	<pre>#include "matrix.h" int mxGetString(const mxArray *pm, char *str, mwSize strlen);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxGetString(pm, str, strlen) mwPointer pm character*(*) str mwSize strlen</pre>
<b>Arguments</b>	<p><b>pm</b> Pointer to a string mxArray; that is, a pointer to an mxArray having the mxCHAR_CLASS class.</p> <p><b>str</b> The starting location into which the string should be written. mxGetString writes the character data into str and then, in C, terminates the string with a NULL character (in the manner of C strings). str can point to either dynamic or static memory.</p> <p><b>strlen</b> Maximum number of characters to read into str. Typically, in C, you set strlen to 1 plus the number of elements in the string mxArray to which pm points. See the mxGetM and mxGetN reference pages to find out how to get the number of elements.</p>
<b>Returns</b>	<p>0 on success, and 1 on failure. Possible reasons for failure include</p> <ul style="list-style-type: none"><li>• Specifying an mxArray that is not a string mxArray.</li><li>• Specifying strlen with less than the number of characters needed to store the entire mxArray pointed to by pm. If this is the case, 1 is returned and the string is truncated.</li></ul>
<b>Description</b>	Call mxGetString to copy the character data of a string mxArray into a C-style string in C or a character array in Fortran. The copied string starts at str and contains no more than strlen-1 characters in C (no

more than `strlen` characters in Fortran). In C, the C-style string is always terminated with a NULL character.

If the string array contains several rows, they are copied—one column at a time—into one long string array.

---

**Note** This function is for use only with strings that represent single-byte character sets. For strings that represent multibyte character sets, use `mxArrayToString`.

---

### C Examples

See `revord.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `explore.c` in the `mex` subdirectory of the `examples` directory
- `mxmalloc.c` in the `mx` subdirectory of the `examples` directory

### See Also

`mxArrayToString`, `mxCreateCharArray`,  
`mxCreateCharMatrixFromStrings`, `mxCreateString`

# mxIsCell (C and Fortran)

---

<b>Purpose</b>	Determine whether input is cell mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsCell(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsCell(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if pm points to an array having the class mxCELL_CLASS, and logical 0 (false) otherwise.
<b>Description</b>	<p>Use mxIsCell to determine whether the specified array is a cell array.</p> <p>In C, calling mxIsCell is equivalent to calling</p> <pre>mxGetClassID(pm) == mxCELL_CLASS</pre> <p>In Fortran, calling mxIsCell is equivalent to calling</p> <pre>mxGetClassName(pm) .eq. 'cell'</pre>
	<hr/> <p><b>Note</b> mxIsCell does not answer the question “Is this mxArray a cell of a cell array?” An individual cell of a cell array can be of any type.</p> <hr/>
<b>See Also</b>	mxIsClass

<b>Purpose</b>	Determine whether input is string mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsChar(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsChar(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if pm points to an array having the class mxCHAR_CLASS, and logical 0 (false) otherwise.
<b>Description</b>	Use mxIsChar to determine whether pm points to string mxArray. In C, calling mxIsChar is equivalent to calling <pre>mxGetClassID(pm) == mxCHAR_CLASS</pre> In Fortran, calling mxIsChar is equivalent to calling <pre>mxGetClassName(pm) .eq. 'char'</pre>
<b>C Examples</b>	See phonebook.c and revord.c in the refbook subdirectory of the examples directory.  For additional examples, see mxcreatecharmatrixfromstr.c, mxislogical.c, and mxmalloc.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxIsClass, mxGetClassID

# mxIsClass (C and Fortran)

---

**Purpose** Determine whether mxArray is member of specified class

**C Syntax**

```
#include "matrix.h"
bool mxIsClass(const mxArray *pm, const char *classname);
```

**Fortran Syntax**

```
integer*4 function mxIsClass(pm, classname)
mwPointer pm
character*(*) classname
```

**Arguments**

pm  
Pointer to an mxArray

classname  
The array category that you are testing. Specify classname as a string (not as an integer identifier). You can specify any one of the following predefined constants:

<b>Value of classname</b>	<b>Corresponding Class</b>
cell	mxCELL_CLASS
char	mxCHAR_CLASS
double	mxDOUBLE_CLASS
function_handle	mxFUNCTION_CLASS
int8	mxINT8_CLASS
int16	mxINT16_CLASS
int32	mxINT32_CLASS
int64	mxINT64_CLASS
logical	mxLOGICAL_CLASS
single	mxSINGLE_CLASS
struct	mxSTRUCT_CLASS
uint8	mxUINT8_CLASS

Value of classname	Corresponding Class
uint16	mxUINT16_CLASS
uint32	mxUINT32_CLASS
uint64	mxUINT64_CLASS
<class_name>	<class_id>
unknown	mxUNKNOWN_CLASS

In the table, <class\_name> represents the name of a specific MATLAB custom object. You can also specify one of your own class names.

## Returns

Logical 1 (true) if pm points to an array having category classname, and logical 0 (false) otherwise.

## Description

Each mxArray is tagged as being a certain type. Call mxIsClass to determine whether the specified mxArray has this type.

In C,

```
mxIsClass("double");
```

is equivalent to calling either of these forms:

```
mxIsDouble(pm);
```

```
strcmp(mxGetClassName(pm), "double");
```

In Fortran,

```
mxIsClass(pm, 'double')
```

is equivalent to calling either one of the following

```
mxIsDouble(pm)
```

```
mxGetClassName(pm) .eq. 'double'
```

## mxIsClass (C and Fortran)

---

It is most efficient to use the `mxIsDouble` form.

### **C Examples**

See `mxisclass.c` in the `mx` subdirectory of the `examples` directory.

### **See Also**

`mxClassID`, `mxGetClassID`, `mxIsEmpty`



<b>Purpose</b>	Determine whether data is complex
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsComplex(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsComplex(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if pm is a numeric array containing complex data, and logical 0 (false) otherwise. If pm points to a cell array or a structure array, mxIsComplex returns false.
<b>Description</b>	Use mxIsComplex to determine whether or not an imaginary part is allocated for an mxArray. The imaginary pointer pi is NULL in C (0L in Fortran) if an mxArray is purely real and does not have any imaginary data. If an mxArray is complex, pi points to an array of numbers.
<b>C Examples</b>	See mxisfinite.c in the mx subdirectory of the examples directory. Additional examples: <ul style="list-style-type: none"><li>• convec.c, phonebook.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory</li><li>• explore.c, yprime.c, mexlock.c, and mexsettrapflag.c in the mex subdirectory of the examples directory</li><li>• mxcalcsinglesubscript.c, mxgeteps.c, and mxgetinf.c in the mx subdirectory of the examples directory</li></ul>
<b>See Also</b>	mxIsNumeric

# mxIsDouble (C and Fortran)

---

<b>Purpose</b>	Determine whether mxArray represents data as double-precision, floating-point numbers
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsDouble(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsDouble(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the mxArray stores its data as double-precision, floating-point numbers, and logical 0 (false) otherwise.
<b>Description</b>	<p>Call <code>mxIsDouble</code> to determine whether or not the specified mxArray represents its real and imaginary data as double-precision, floating-point numbers.</p> <p>Older versions of MATLAB store all mxArray data as double-precision, floating-point numbers. However, starting with MATLAB Version 5, MATLAB can store real and imaginary data in a variety of numerical formats.</p> <p>In C, calling <code>mxIsDouble</code> is equivalent to calling</p> <pre>mxGetClassID(pm) == mxDOUBLE_CLASS</pre> <p>In Fortran, calling <code>mxIsDouble</code> is equivalent to calling</p> <pre>mxGetClassName(pm) .eq. 'double'</pre>
<b>C Examples</b>	<p>See <code>findnz.c</code>, <code>fulltosparse.c</code>, <code>timestwo.c</code>, and <code>xtimesy.c</code> in the <code>refbook</code> subdirectory of the examples directory.</p> <p>Additional examples:</p>

- `mexget.c`, `mexlock.c`, `mexsettrapflag.c`, and `yprime.c` in the `mex` subdirectory of the `examples` directory
- `mxcalcsinglesubscript.c`, `mxgeteps.c`, `mxgetinf.c`, and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory

### See Also

`mxIsClass`, `mxGetClassID`

## mxIsEmpty (C andFortran)

---

<b>Purpose</b>	Determine whether mxArray is empty
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsEmpty(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsEmpty(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the mxArray is empty, and logical 0 (false) otherwise.
<b>Description</b>	Use mxIsEmpty to determine whether an mxArray contains no data. An mxArray is empty if the size of any of its dimensions is 0.
<b>C Examples</b>	See mxisfinite.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxIsClass

<b>Purpose</b>	Determine whether input is finite
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsFinite(double value);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsFinite(value) real*8 value</pre>
<b>Arguments</b>	value The double-precision, floating-point number that you are testing
<b>Returns</b>	Logical 1 (true) if value is finite, and logical 0 (false) otherwise.
<b>Description</b>	Call <code>mxIsFinite</code> to determine whether or not value is finite. A number is finite if it is greater than <code>-Inf</code> and less than <code>Inf</code> .
<b>C Examples</b>	See <code>mxisfinite.c</code> in the <code>mx</code> subdirectory of the <code>examples</code> directory.
<b>See Also</b>	<code>mxIsInf</code> , <code>mxIsNan</code>

# mxIsFromGlobalWS (C and Fortran)

---

<b>Purpose</b>	Determine whether mxArray was copied from MATLAB global workspace
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsFromGlobalWS(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsFromGlobalWS(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the array was copied out of the global workspace, and logical 0 (false) otherwise.
<b>Description</b>	mxIsFromGlobalWS is useful for stand-alone MAT programs. mexIsGlobal tells you whether the pointer you pass actually points into the global workspace.
<b>C Examples</b>	See matdgn.c and matcreat.c in the eng_mat subdirectory of the examples directory.
<b>See Also</b>	mexIsGlobal

<b>Purpose</b>	Determine whether input is infinite
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsInf(double value);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsInf(value) real*8 value</pre>
<b>Arguments</b>	<p>value</p> <p>The double-precision, floating-point number that you are testing</p>
<b>Returns</b>	Logical 1 (true) if value is infinite, and logical 0 (false) otherwise.
<b>Description</b>	<p>Call <code>mxIsInf</code> to determine whether or not value is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named <code>Inf</code>, which represents IEEE arithmetic positive infinity. The value of the variable <code>Inf</code> is built into the system; you cannot modify it.</p> <p>Operations that return infinity include</p> <ul style="list-style-type: none"><li>• Division by 0. For example, <code>5/0</code> returns infinity.</li><li>• Operations resulting in overflow. For example, <code>exp(10000)</code> returns infinity because the result is too large to be represented on your machine.</li></ul> <p>If value equals NaN (Not-a-Number), <code>mxIsInf</code> returns false. In other words, NaN is not equal to infinity.</p>
<b>C Examples</b>	See <code>mxisfinite.c</code> in the <code>mx</code> subdirectory of the examples directory.
<b>See Also</b>	<code>mxIsFinite</code> , <code>mxIsNaN</code>

## mxIsInt16 (C and Fortran)

---

**Purpose** Determine whether mxArray represents data as signed 16-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsInt16(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsInt16(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the array stores its data as signed 16-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsInt16 to determine whether or not the specified array represents its real and imaginary data as 16-bit signed integers.

In C, calling mxIsInt16 is equivalent to calling

```
mxGetClassID(pm) == mxINT16_CLASS
```

In Fortran, calling mxIsInt16 is equivalent to calling

```
mxGetClassName(pm) == 'int16'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt32, mxIsInt64, mxIsUInt8, mxIsUInt16, mxIsUInt32, mxIsUInt64



**Purpose** Determine whether mxArray represents data as signed 32-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsInt32(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsInt32(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the array stores its data as signed 32-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsInt32 to determine whether or not the specified array represents its real and imaginary data as 32-bit signed integers.

In C, calling mxIsInt32 is equivalent to calling

```
mxGetClassID(pm) == mxINT32_CLASS
```

In Fortran, calling mxIsInt32 is equivalent to calling

```
mxGetClassName(pm) == 'int32'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

## mxIsInt64 (C and Fortran)

---

**Purpose** Determine whether mxArray represents data as signed 64-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsInt64(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsInt64(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the array stores its data as signed 64-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsInt64 to determine whether or not the specified array represents its real and imaginary data as 64-bit signed integers.

In C, calling mxIsInt64 is equivalent to calling

```
mxGetClassID(pm) == mxINT64_CLASS
```

In Fortran, calling mxIsInt64 is equivalent to calling

```
mxGetClassName(pm) == 'int64'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

**Purpose** Determine whether mxArray represents data as signed 8-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsInt8(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsInt8(pm)
mwPointer pm
```

**Arguments**

pm  
Pointer to an mxArray

**Returns** Logical 1 (true) if the array stores its data as signed 8-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsInt8 to determine whether or not the specified array represents its real and imaginary data as 8-bit signed integers.

In C, calling mxIsInt8 is equivalent to calling

```
mxGetClassID(pm) == mxINT8_CLASS
```

In Fortran, calling mxIsInt8 is equivalent to calling

```
mxGetClassName(pm) .eq. 'int8'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUInt8, mxIsUInt16, mxIsUInt32, mxIsUInt64

# mxIsLogical (C and Fortran)

---

<b>Purpose</b>	Determine whether mxArray is of class mxLogical
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsLogical(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsLogical(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if pm points to a logical mxArray, and logical 0 (false) otherwise.
<b>Description</b>	Use mxIsLogical to determine whether MATLAB treats the data in the mxArray as Boolean (logical). If an mxArray is logical, MATLAB treats all zeros as meaning false and all nonzero values as meaning true. For additional information on the use of logical variables in MATLAB, type help logical at the MATLAB prompt.
<b>C Examples</b>	See mxislogical.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxIsClass

<b>Purpose</b>	Determine whether scalar mxArray is of class mxLogical
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsLogicalScalar(const mxArray *array_ptr);</pre>
<b>Arguments</b>	array_ptr Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the mxArray is of class mxLogical and has 1-by-1 dimensions, and logical 0 (false) otherwise.
<b>Description</b>	<p>Use mxIsLogicalScalar to determine whether MATLAB treats the scalar data in the mxArray as logical or numerical. For additional information on the use of logical variables in MATLAB, type help logical at the MATLAB prompt.</p> <p>mxIsLogicalScalar(pa) is equivalent to</p> <pre>mxIsLogical(pa) &amp;&amp; mxGetNumberOfElements(pa) == 1</pre>
<b>See Also</b>	mxIsLogical, mxIsLogicalScalarTrue, mxGetLogicals, mxGetScalar

## mxIsLogicalScalarTrue (C)

---

<b>Purpose</b>	Determine whether scalar mxArray of class mxLogical is true
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsLogicalScalarTrue(const mxArray *array_ptr);</pre>
<b>Arguments</b>	array_ptr Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the value of the mxArray's logical, scalar element is true, and logical 0 (false) otherwise.
<b>Description</b>	<p>Use mxIsLogicalScalarTrue to determine whether the value of a scalar mxArray is true or false. For additional information on the use of logical variables in MATLAB, type help logical at the MATLAB prompt.</p> <p>mxIsLogicalScalarTrue(pa) is equivalent to</p> <pre>mxIsLogical(pa) &amp;&amp; mxGetNumberOfElements(pa) == 1 &amp;&amp; mxGetLogicals(pa)[0] == true</pre>
<b>See Also</b>	mxIsLogical, mxIsLogicalScalar, mxGetLogicals, mxGetScalar

<b>Purpose</b>	Determine whether input is NaN (Not-a-Number)
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsNaN(double value);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsNaN(value) real*8 value</pre>
<b>Arguments</b>	<p>value</p> <p>The double-precision, floating-point number that you are testing</p>
<b>Returns</b>	Logical 1 (true) if value is NaN (Not-a-Number), and logical 0 (false) otherwise.
<b>Description</b>	<p>Call <code>mxIsNaN</code> to determine whether or not <code>value</code> is NaN. NaN is the IEEE arithmetic representation for Not-a-Number. A NaN is obtained as a result of mathematically undefined operations such as</p> <ul style="list-style-type: none"><li>• <code>0.0/0.0</code></li><li>• <code>Inf - Inf</code></li></ul> <p>The system understands a family of bit patterns as representing NaN. In other words, NaN is not a single value; rather, it is a family of numbers that MATLAB (and other IEEE-compliant applications) use to represent an error condition or missing data.</p>
<b>C Examples</b>	<p>See <code>mxIsFinite.c</code> in the <code>mx</code> subdirectory of the examples directory.</p> <p>For additional examples, see <code>findnz.c</code> and <code>fulltosparse.c</code> in the <code>refbook</code> subdirectory of the examples directory.</p>
<b>See Also</b>	<code>mxIsFinite</code> , <code>mxIsInf</code>

# mxIsNumeric (C and Fortran)

---

<b>Purpose</b>	Determine whether mxArray is numeric
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsNumeric(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsNumeric(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the array can contain numeric data. The following class IDs represent storage types for arrays that can contain numeric data: <ul style="list-style-type: none"><li>• mxDOUBLE_CLASS</li><li>• mxSINGLE_CLASS</li><li>• mxINT8_CLASS</li><li>• mxUINT8_CLASS</li><li>• mxINT16_CLASS</li><li>• mxUINT16_CLASS</li><li>• mxINT32_CLASS</li><li>• mxUINT32_CLASS</li><li>• mxINT64_CLASS</li><li>• mxUINT64_CLASS</li></ul> Logical 0 (false) if the array cannot contain numeric data.
<b>Description</b>	Call mxIsNumeric to determine whether the specified array contains numeric data. If the specified array has a storage type that represents



numeric data, `mxIsNumeric` returns logical 1 (true). Otherwise, `mxIsNumeric` returns logical 0 (false).

Call `mxGetClassID` to determine the exact storage type.

### **C Examples**

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

### **Fortran Examples**

See `matdemo1.F` in the `eng_mat` subdirectory of the `examples` directory.

### **See Also**

`mxGetClassID`

# mxIsSingle (C and Fortran)

---

<b>Purpose</b>	Determine whether mxArray represents data as single-precision, floating-point numbers
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsSingle(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsSingle(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if the array stores its data as single-precision, floating-point numbers, and logical 0 (false) otherwise.
<b>Description</b>	<p>Use <code>mxIsSingle</code> to determine whether or not the specified array represents its real and imaginary data as single-precision, floating-point numbers.</p> <p>In C, calling <code>mxIsSingle</code> is equivalent to calling</p> <pre>mxGetClassID(pm) == mxSINGLE_CLASS</pre> <p>In Fortran, calling <code>mxIsSingle</code> is equivalent to calling</p> <pre>mxGetClassName(pm) .eq. 'single'</pre>
<b>See Also</b>	<code>mxIsClass</code> , <code>mxGetClassID</code>

<b>Purpose</b>	Determine whether input is sparse mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsSparse(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsSparse(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if pm points to a sparse mxArray, and logical 0 (false) otherwise. A false return value means that pm points to a full mxArray or that pm does not point to a legal mxArray.
<b>Description</b>	Use mxIsSparse to determine whether pm points to a sparse mxArray. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.
<b>C Examples</b>	See phonebook.c in the refbook subdirectory of the examples directory. For additional examples, see mxgetnzmax.c, mxsetdimensions.c, and mxsetnzmax.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxGetIr, mxGetJc, mxCreateSparse

# mxIsStruct (C and Fortran)

---

<b>Purpose</b>	Determine whether input is structure mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" bool mxIsStruct(const mxArray *pm);</pre>
<b>Fortran Syntax</b>	<pre>integer*4 function mxIsStruct(pm) mwPointer pm</pre>
<b>Arguments</b>	pm Pointer to an mxArray
<b>Returns</b>	Logical 1 (true) if pm points to a structure mxArray, and logical 0 (false) otherwise.
<b>Description</b>	Use mxIsStruct to determine whether pm points to a structure mxArray. Many routines (for example, mxGetFieldName and mxSetField) require a structure mxArray as an argument.
<b>C Examples</b>	See phonebook.c in the refbook subdirectory of the examples directory.
<b>See Also</b>	mxCreateStructArray, mxCreateStructMatrix, mxGetNumberOfFields, mxGetField, mxSetField

**Purpose** Determine whether mxArray represents data as unsigned 16-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsUint16(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsUint16(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the mxArray stores its data as unsigned 16-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsUint16 to determine whether or not the specified mxArray represents its real and imaginary data as 16-bit unsigned integers.

In C, calling mxIsUint16 is equivalent to calling

```
mxGetClassID(pm) == mxUINT16_CLASS
```

In Fortran, calling mxIsUint16 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint16'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint32, mxIsUint64

## mxIsUint32 (C and Fortran)

---

**Purpose** Determine whether mxArray represents data as unsigned 32-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsUint32(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsUint32(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the mxArray stores its data as unsigned 32-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsUint32 to determine whether or not the specified mxArray represents its real and imaginary data as 32-bit unsigned integers.

In C, calling mxIsUint32 is equivalent to calling

```
mxGetClassID(pm) == mxUINT32_CLASS
```

In Fortran, calling mxIsUint32 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint32'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint64

**Purpose** Determine whether mxArray represents data as unsigned 64-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsUint64(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsUint64(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the mxArray stores its data as unsigned 64-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsUint64 to determine whether or not the specified mxArray represents its real and imaginary data as 64-bit unsigned integers.

In C, calling mxIsUint64 is equivalent to calling

```
mxGetClassID(pm) == mxUINT64_CLASS
```

In Fortran, calling mxIsUint64 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint64'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32

# mxIsUint8 (C and Fortran)

---

**Purpose** Determine whether mxArray represents data as unsigned 8-bit integers

**C Syntax**

```
#include "matrix.h"
bool mxIsUint8(const mxArray *pm);
```

**Fortran Syntax**

```
integer*4 function mxIsUint8(pm)
mwPointer pm
```

**Arguments**

pm  
    Pointer to an mxArray

**Returns** Logical 1 (true) if the mxArray stores its data as unsigned 8-bit integers, and logical 0 (false) otherwise.

**Description** Use mxIsUint8 to determine whether or not the specified mxArray represents its real and imaginary data as 8-bit unsigned integers.

In C, calling mxIsUint8 is equivalent to calling

```
mxGetClassID(pm) == mxUINT8_CLASS
```

In Fortran, calling mxIsUint8 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint8'
```

**See Also** mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint16, mxIsUint32, mxIsUint64



<b>Purpose</b>	Allocate dynamic memory using MATLAB memory manager
<b>C Syntax</b>	<pre>#include "matrix.h" #include &lt;stdlib.h&gt; void *mxMalloc(size_t n);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxMalloc(n) mwSize n</pre>
<b>Arguments</b>	<p>n</p> <p>Number of bytes to allocate</p>
<b>Returns</b>	<p>A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, <code>mxMalloc</code> returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.</p> <p><code>mxMalloc</code> is unsuccessful when there is insufficient free heap space.</p>
<b>Description</b>	<p>MATLAB applications should always call <code>mxMalloc</code> rather than <code>malloc</code> to allocate memory.</p> <p><code>mxMalloc</code> works differently in MEX-files than in stand-alone MATLAB applications. In MEX-files, <code>mxMalloc</code> automatically</p> <ul style="list-style-type: none"><li>• Allocates enough contiguous heap space to hold <code>n</code> bytes.</li><li>• Registers the returned heap space with the MATLAB memory management facility.</li></ul> <p>The MATLAB memory management facility maintains a list of all memory allocated by <code>mxMalloc</code>. The MATLAB memory management facility automatically frees (deallocates) all of a MEX-file's parcels when control returns to the MATLAB prompt.</p> <p>In stand-alone MATLAB C applications, <code>mxMalloc</code> calls the ANSI C <code>malloc</code> function.</p>

## mxMalloc (C and Fortran)

---

By default, in a MEX-file, `mxMalloc` generates nonpersistent `mxMalloc` data. In other words, the memory management facility automatically deallocates the memory as soon as the MEX-file ends. If you want the memory to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxMalloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX-file is cleared.

When you finish using the memory allocated by `mxMalloc`, call `mxFree`. `mxFree` deallocates the memory.

### **C Examples**

See `mxmalloc.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

### **See Also**

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxFree`, `mxRealloc`

<b>Purpose</b>	Reallocate memory
<b>C Syntax</b>	<pre>#include "matrix.h" #include &lt;stdlib.h&gt; void *mxRealloc(void *ptr, size_t size);</pre>
<b>Fortran Syntax</b>	<pre>mwPointer function mxRealloc(ptr, size) mwPointer ptr mwSize size</pre>
<b>Arguments</b>	<p><b>ptr</b> Pointer to a block of memory allocated by mxCalloc, mxMalloc, or mxRealloc</p> <p><b>size</b> New size of allocated memory, in bytes</p>
<b>Returns</b>	A pointer to the reallocated block of memory, or NULL in C (0 in Fortran) if size is 0. In a stand-alone (non-MEX-file) application, if not enough memory is available to expand the block to the given size, mxRealloc returns NULL in C (0 in Fortran). In a MEX-file, if not enough memory is available to expand the block to the given size, the MEX-file terminates and control returns to the MATLAB prompt.
<b>Description</b>	<p>mxRealloc changes the size of a memory block that has been allocated with mxCalloc, mxMalloc, or mxRealloc.</p> <p>If size is 0 and ptr is not NULL in C (0 in Fortran), mxRealloc frees the memory pointed to by ptr and returns NULL in C (0 in Fortran).</p> <p>If size is greater than 0 and ptr is NULL in C (0 in Fortran), mxRealloc behaves like mxMalloc, allocating a new block of memory of size bytes and returning a pointer to the new block.</p> <p>Otherwise, mxRealloc changes the size of the memory block pointed to by ptr to size bytes. The contents of the reallocated memory are unchanged up to the smaller of the new and old sizes. The reallocated memory may be in a different location from the original memory, so</p>

## mxRealloc (C and Fortran)

---

the returned pointer can be different from `ptr`. If the memory location changes, `mxRealloc` frees the original memory block pointed to by `ptr`.

In a stand-alone (non-MEX-file) application, if not enough memory is available to expand the block to the given size, `mxRealloc` returns `NULL` in C (0 in Fortran) and leaves the original memory block unchanged. You must use `mxFree` to free the original memory block.

MATLAB maintains a list of all memory allocated by `mxRealloc`. By default, in a MEX-file, `mxRealloc` generates nonpersistent `mxRealloc` data. The memory management facility automatically deallocates the memory as soon as the MEX-file ends.

If you want the memory to persist after a MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxRealloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory when your MEX-file is cleared.

When you finish using the memory allocated by `mxRealloc`, call `mxFree`. `mxFree` deallocates the memory.

### **C Examples**

See `mxsetnzmax.c` in the `mx` subdirectory of the examples directory.

### **See Also**

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxMalloc`, `mxDestroyArray`, `mxFree`, `mxMalloc`

<b>Purpose</b>	Remove field from structure array
<b>C Syntax</b>	<pre>#include "matrix.h" extern void mxRemoveField(mxArray pm, int fieldnumber);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxRemoveField(pm, fieldnumber) mwPointer pm integer*4 fieldnumber</pre>
<b>Arguments</b>	<p>pm     Pointer to a structure mxArray</p> <p>fieldnumber     The number of the field you want to remove. In C, to remove the first field, set fieldnumber to 0; to remove the second field, set fieldnumber to 1; and so on. In Fortran, to remove the first field, set fieldnumber to 1; to remove the second field, set fieldnumber to 2; and so on.</p>
<b>Description</b>	<p>Call <code>mxRemoveField</code> to remove a field from a structure array. If the field does not exist, nothing happens. This function does not destroy the field values. Use <code>mxDestroyArray</code> to destroy the actual field values.</p> <p>Consider a MATLAB structure initialized to</p> <pre>patient.name = 'John Doe'; patient.billing = 127.00; patient.test = [79 75 73; 180 178 177.5; 220 210 205];</pre> <p>In C, the field number 0 represents the field name; field number 1 represents field <code>billing</code>; field number 2 represents field <code>test</code>. In Fortran, the field number 1 represents the field name; field number 2 represents field <code>billing</code>; field number 3 represents field <code>test</code>.</p>
<b>See Also</b>	<code>mxAddField</code> , <code>mxDestroyArray</code> , <code>mxGetFieldByNumber</code>

# mxSetCell (C and Fortran)

---

**Purpose** Set value of one cell of mxArray

**C Syntax**

```
#include "matrix.h"
void mxSetCell(mxArray *pm, mwIndex index, mxArray *value);
```

**Fortran Syntax**

```
subroutine mxSetCell(pm, index, value)
mwPointer pm, value
mwIndex index
```

**Arguments**

pm  
Pointer to a cell mxArray

index  
Index from the beginning of the mxArray. Specify the number of elements between the first cell of the mxArray and the cell you want to set. The easiest way to calculate index in a multidimensional cell array is to call mxCalcSingleSubscript.

value  
The new value of the cell. You can put any kind of mxArray into a cell. In fact, you can even put another cell mxArray into a cell.

**Description** Call mxSetCell to put the designated value into a particular cell of a cell mxArray.

---

**Note** Inputs to a MEX-file are constant read-only mxArrays and should not be modified. Using mxSetCell\* or mxSetField\* to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

---

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetCell before you call mxSetCell.

### **C Examples**

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory. For an additional example, see `mxcreatecellmatrix.c` in the `mx` subdirectory of the `examples` directory.

### **See Also**

`mxCreateCellArray`, `mxCreateCellMatrix`, `mxGetCell`, `mxIsCell`, `mxFree`

## mxSetClassName (C)

---

<b>Purpose</b>	Convert structure array to MATLAB object array
<b>C Syntax</b>	<pre>#include "matrix.h" int mxSetClassName(mxArray *array_ptr, const char *classname);</pre>
<b>Arguments</b>	<pre>array_ptr     Pointer to an mxArray of class mxSTRUCT_CLASS classname     The object class to which to convert array_ptr</pre>
<b>Returns</b>	0 if successful, and nonzero otherwise.
<b>Description</b>	mxSetClassName converts a structure array to an object array, to be saved subsequently to a MAT-file. The object is not registered or validated by MATLAB until it is loaded via the LOAD command. If the specified classname is an undefined class within MATLAB, LOAD converts the object back to a simple structure array.
<b>See Also</b>	mxIsClass, mxGetClassID



<b>Purpose</b>	Set pointer to data
<b>C Syntax</b>	<pre>#include "matrix.h" void mxSetData(mxArray *pm, void *pr);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxSetData(pm, pr) mwPointer pm, pr</pre>
<b>Arguments</b>	<p>pm Pointer to an mxArray</p> <p>pr Pointer to an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call mxCalloc to allocate this memory.</p>
<b>Description</b>	<p>mxSetData is similar to mxSetPr, except that in C, its second argument is a void *. Use this on numeric arrays with contents other than double.</p> <p>This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetData before you call mxSetData.</p>
<b>See Also</b>	mxCalloc, mxFree, mxGetData, mxSetPr

# mxSetDimensions (C and Fortran)

---

**Purpose** Modify number of dimensions and size of each dimension

**C Syntax**

```
#include "matrix.h"
int mxSetDimensions(mxArray *pm, const mwSize *dims,
    mwSize ndim);
```

**Fortran Syntax**

```
integer*4 function mxSetDimensions(pm, dims, ndim)
mwPointer pm
mwSize dims, ndim
```

**Arguments**

pm  
Pointer to an mxArray

dims  
The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to 5 and `dims[1]` to 7 establishes a 5-by-7 mxArray. In Fortran, setting `dims(1)` to 5 and `dims(2)` to 7 establishes a 5-by-7 mxArray. In most cases, there should be `ndim` elements in the `dims` array.

ndim  
The desired number of dimensions

**Returns** 0 on success, and 1 on failure. `mxSetDimensions` allocates heap space to hold the input size array. So it is possible (though extremely unlikely) that increasing the number of dimensions can cause the system to run out of heap space.

**Description** Call `mxSetDimensions` to reshape an existing mxArray. `mxSetDimensions` is similar to `mxSetM` and `mxSetN`; however, `mxSetDimensions` provides greater control for reshaping mxArrays that have more than two dimensions.

`mxSetDimensions` does not allocate or deallocate any space for the `pr` or `pi` arrays. Consequently, if your call to `mxSetDimensions` increases the number of elements in the mxArray, you must enlarge the `pr` (and `pi`, if it exists) arrays accordingly.

If your call to `mxSetDimensions` reduces the number of elements in the `mxArray`, you can optionally reduce the size of the `pr` and `pi` arrays using `mxRealloc`.

Any trailing singleton dimensions specified in the `dims` argument are automatically removed from the resulting array. For example, if `ndim` equals 5 and `dims` equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.

### **C Examples**

See `mxsetdimensions.c` in the `mx` subdirectory of the examples directory.

### **See Also**

`mxGetNumberOfDimensions`, `mxSetM`, `mxSetN`, `mxRealloc`

# mxSetField (C and Fortran)

---

**Purpose** Set structure array field, given field name and index

**C Syntax**

```
#include "matrix.h"
void mxSetField(mxArray *pm, mwIndex index,
    const char *fieldname, mxArray *value);
```

**Fortran Syntax**

```
subroutine mxSetField(pm, index, fieldname, value)
mwPointer pm, value
mwIndex index
character*(*) fieldname
```

**Arguments**

**pm**  
Pointer to a structure mxArray. Call `mxIsStruct` to determine whether `pm` points to a structure mxArray.

**index**  
Index of the desired element. In C, the first element of an mxArray has an index of 0, the second element has an index of 1, and the last element has an index of  $N-1$ , where  $N$  is the total number of elements in the structure mxArray. In Fortran, the first element of an mxArray has an index of 1, the second element has an index of 2, and the last element has an index of  $N$ . See `mxCalcSingleSubscript` for details on calculating an index.

**fieldname**  
The name of the field whose value you are assigning. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to determine existing field names.

**value**  
Pointer to the mxArray you are assigning.

**Description** Use `mxSetField` to assign a value to the specified element of the specified field. In pseudo-C terminology, `mxSetField` performs the assignment

```
pm[index].fieldname = value;
```

---

**Note** Inputs to a MEX-file are constant read-only mxArray's and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

---

In C, calling

```
mxSetField(pa, index, "fieldname", new_value_pa);
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "fieldname");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

In Fortran, calling

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetField` before you call `mxSetField`.

## C Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the examples directory.

## See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetFieldByNumber`, `mxFree`

# mxSetFieldByNumber (C and Fortran)

---

**Purpose** Set structure array field, given field number and index

**C Syntax**

```
#include "matrix.h"
void mxSetFieldByNumber(mxArray *pm, mwIndex index,
    int fieldnumber, mxArray *value);
```

**Fortran Syntax**

```
subroutine mxSetFieldByNumber(pm, index, fieldnumber, value)
mwPointer pm, value
mwIndex index
integer*4 fieldnumber
```

**Arguments**

**pm** Pointer to a structure mxArray. Call `mxIsStruct` to determine whether `pm` points to a structure mxArray.

**index** The desired element. In C, the first element of an mxArray has an index of 0, the second element has an index of 1, and the last element has an index of `N-1`, where `N` is the total number of elements in the structure mxArray. In Fortran, the first element of an mxArray has an index of 1, the second element has an index of 2, and the last element has an index of `N`. See `mxCalcSingleSubscript` for details on calculating an index.

**fieldnumber** The position of the field whose value you want to extract. In C, the first field within each element has a `fieldnumber` of 0, the second field has a `fieldnumber` of 1, and so on. The last field has a `fieldnumber` of `N-1`, where `N` is the number of fields. In Fortran, the first field within each element has a `fieldnumber` of 1, the second field has a `fieldnumber` of 2, and so on. The last field has a `fieldnumber` of `N`.

**value** The value you are assigning.

**Description** Use `mxSetFieldByNumber` to assign a value to the specified element of the specified field. `mxSetFieldByNumber` is almost identical to

# mxSetFieldByNumber (C and Fortran)

---

`mxSetField`; however, the former takes a field number as its third argument and the latter takes a field name as its third argument.

---

**Note** Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

---

In C, calling

```
mxSetField(pa, index, "field_name", new_value_pa);
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

In Fortran, calling

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetFieldByNumber` before you call `mxSetFieldByNumber`.

## C Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

## mxSetFieldByNumber (C and Fortran)

---

### **See Also**

mxCreateStructArray, mxCreateStructMatrix, mxGetField,  
mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber,  
mxGetNumberOfFields, mxIsStruct, mxSetField, mxFree



# mxSetImagData (C and Fortran)

---

<b>Purpose</b>	Set imaginary data pointer for mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" void mxSetImagData(mxArray *pm, void *pi);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxSetImagData(pm, pi) mwPointer pm, pi</pre>
<b>Arguments</b>	<p><b>pm</b> Pointer to an mxArray</p> <p><b>pi</b> Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pi points to static memory, memory errors will result when the array is destroyed.</p>
<b>Description</b>	<p>mxSetImagData is similar to mxSetPi, except that in C, its pi argument is a void *. Use this on numeric arrays with contents other than double.</p> <p>This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetImagData before you call mxSetImagData.</p>
<b>C Examples</b>	See mxisfinite.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxCalloc, mxFree, mxGetImagData, mxSetPi

# mxSetIr (C and Fortran)

---

**Purpose** Set `ir` array of sparse `mxArray`

**C Syntax**

```
#include "matrix.h"
void mxSetIr(mxArray *pm, mwIndex *ir);
```

**Fortran Syntax**

```
subroutine mxSetIr(pm, ir)
mwPointer pm, ir
```

**Arguments**

`pm`  
Pointer to a sparse `mxArray`

`ir`  
Pointer to the `ir` array. The `ir` array must be sorted in column-major order.

**Description** Use `mxSetIr` to specify the `ir` array of a sparse `mxArray`. The `ir` array is an array of integers; the length of the `ir` array should equal the value of `nzmax`.

Each element in the `ir` array indicates a row (offset by 1) at which a nonzero element can be found. (The `jc` array is an index that indirectly specifies a column where nonzero elements can be found. See `mxSetJc` for more details on `jc`.)

For example, suppose you create a 7-by-3 sparse `mxArray` named `Sparrow` containing six nonzero elements by typing

```
Sparrow = zeros(7,3);
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The `pr` array holds the real data for the sparse matrix, which in Sparrow is the five 1s and the one 2. If there is any nonzero imaginary data, it is in a `pi` array.

Subscript	ir	pr	jc	Comments
(2,1)	1	1	0	Column 1; ir is 1 because row is 2.
(5,1)	4	1	2	Column 1; ir is 4 because row is 5.
(3,2)	2	1	3	Column 2; ir is 2 because row is 3.
(2,3)	1	2	6	Column 3; ir is 1 because row is 2.
(5,3)	4	1		Column 3; ir is 4 because row is 5.
(6,3)	5	1		Column 3; ir is 5 because row is 6.

Notice how each element of the `ir` array is always 1 less than the row of the corresponding nonzero element. For instance, the first nonzero element is in row 2; therefore, the first element in `ir` is 1 (that is,  $2 - 1$ ). The second nonzero element is in row 5; therefore, the second element in `ir` is 4 ( $5 - 1$ ).

The `ir` array must be in column-major order. That means that the `ir` array must define the row positions in column 1 (if any) first, then the row positions in column 2 (if any) second, and so on through column  $N$ . Within each column, row position 1 must appear prior to row position 2, and so on.

`mxSetIr` does not sort the `ir` array for you; you must specify an `ir` array that is already sorted.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetIr` before you call `mxSetIr`.

## C Examples

See `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `explore.c` in the `mex` subdirectory of the `examples` directory.

## mxSetIr (C and Fortran)

---

### **See Also**

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetJc`, `mxFree`

<b>Purpose</b>	Set jc array of sparse mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" void mxSetJc(mxArray *pm, mwIndex *jc);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxSetJc(pm, jc) mwPointer pm, jc</pre>
<b>Arguments</b>	<p>pm     Pointer to a sparse mxArray</p> <p>jc     Pointer to the jc array</p>
<b>Description</b>	<p>Use mxSetJc to specify a new jc array for a sparse mxArray. The jc array is an integer array having n+1 elements, where n is the number of columns in the sparse mxArray.</p> <p>If the jth column of the sparse mxArray has any nonzero elements:</p> <ul style="list-style-type: none"><li>• jc[j] is the index in ir, pr, and pi (if it exists) of the first nonzero element in the jth column.</li><li>• jc[j+1]-1 is the index of the last nonzero element in the jth column.</li></ul> <p>The number of nonzero elements in the jth column of the sparse mxArray is</p> $jc[j+1] - jc[j];$ <p>For the jth column of the sparse mxArray, jc[j] is the total number of nonzero elements in all preceding columns. The last element of the jc array, jc[number of columns], is equal to nnz, which is the number of nonzero elements in the entire sparse mxArray.</p> <p>For example, consider a 7-by-3 sparse mxArray named Sparrow containing six nonzero elements, created by typing</p> <pre>Sparrow = zeros(7,3);</pre>

## mxSetJc (C and Fortran)

---

```
Sparrow(2,1) = 1;  
Sparrow(5,1) = 1;  
Sparrow(3,2) = 1;  
Sparrow(2,3) = 2;  
Sparrow(5,3) = 1;  
Sparrow(6,3) = 1;  
Sparrow = sparse(Sparrow);
```

The contents of the `ir`, `jc`, and `pr` arrays are listed in this table.

Subscript	ir	pr	jc	Comment
(2,1)	1	1	0	Column 1 contains two nonzero elements, with rows designated by <code>ir[0]</code> and <code>ir[1]</code>
(5,1)	4	1	2	Column 2 contains one nonzero element, with row designated by <code>ir[2]</code>
(3,2)	2	1	3	Column 3 contains three nonzero elements, with rows designated by <code>ir[3]</code> , <code>ir[4]</code> , and <code>ir[5]</code>
(2,3)	1	2	6	There are six nonzero elements in all.
(5,3)	4	1		
(6,3)	5	1		

As an example of a much sparser `mxArray`, consider a 1,000-by-8 sparse `mxArray` named `Spacious` containing only three nonzero elements. The `ir`, `pr`, and `jc` arrays contain the values listed in this table.

Subscript	ir	pr	jc	Comment
(73,2)	72	1	0	Column 1 contains no nonzero elements.
(50,3)	49	1	0	Column 2 contains one nonzero element, with row designated by ir[0].
(64,5)	63	1	1	Column 3 contains one nonzero element, with row designated by ir[1].
			2	Column 4 contains no nonzero elements.
			2	Column 5 contains one nonzero element, with row designated by ir[2].
			3	Column 6 contains no nonzero elements.
			3	Column 7 contains no nonzero elements.
			3	Column 8 contains no nonzero elements.
			3	There are three nonzero elements in all.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetJc` before you call `mxSetJc`.

## C Examples

See `mxsetdimensions.c` in the `mx` subdirectory of the examples directory. For an additional example, see `explore.c` in the `mex` subdirectory of the examples directory.

## See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetIr`, `mxFree`

# mxSetM (C and Fortran)

---

**Purpose** Set number of rows in mxArray

**C Syntax**

```
#include "matrix.h"
void mxSetM(mxArray *pm, mwSize m);
```

**Fortran Syntax**

```
subroutine mxSetM(pm, m)
mwPointer pm
mwSize m
```

**Arguments**

pm  
    Pointer to an mxArray

m  
    The desired number of rows

**Description** Call mxSetM to set the number of rows in the specified mxArray. The term “rows” means the first dimension of an mxArray, regardless of the number of dimensions. Call mxSetN to set the number of columns.

You typically use mxSetM to change the shape of an existing mxArray. Note that mxSetM does not allocate or deallocate any space for the pr, pi, ir, or jc arrays. Consequently, if your calls to mxSetM and mxSetN increase the number of elements in the mxArray, you must enlarge the pr, pi, ir, and/or jc arrays. Call mxRealloc to enlarge them.

If your calls to mxSetM and mxSetN end up reducing the number of elements in the mxArray, you may want to reduce the sizes of the pr, pi, ir, and/or jc arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

**C Examples** See mxsetdimensions.c in the mx subdirectory of the examples directory. For an additional example, see sincall.c in the refbook subdirectory of the examples directory.

**See Also** mxGetM, mxGetN, mxSetN



<b>Purpose</b>	Set number of columns in mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" void mxSetN(mxArray *pm, mwSize n);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxSetN(pm, n) mwPointer pm mwSize n</pre>
<b>Arguments</b>	<p>pm     Pointer to an mxArray</p> <p>n     The desired number of columns</p>
<b>Description</b>	<p>Call mxSetN to set the number of columns in the specified mxArray. The term “columns” always means the second dimension of a matrix. Calling mxSetN forces an mxArray to have two dimensions. For example, if pm points to an mxArray having three dimensions, calling mxSetN reduces the mxArray to two dimensions.</p> <p>You typically use mxSetN to change the shape of an existing mxArray. Note that mxSetN does not allocate or deallocate any space for the pr, pi, ir, or jc arrays. Consequently, if your calls to mxSetN and mxSetM increase the number of elements in the mxArray, you must enlarge the pr, pi, ir, and/or jc arrays.</p> <p>If your calls to mxSetM and mxSetN end up reducing the number of elements in the mxArray, you may want to reduce the sizes of the pr, pi, ir, and/or jc arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.</p>
<b>C Examples</b>	See mxsetdimensions.c in the mx subdirectory of the examples directory. For an additional example, see sincall.c in the refbook subdirectory of the examples directory.
<b>See Also</b>	mxGetM, mxGetN, mxSetM

# mxSetNzmax (C and Fortran)

---

**Purpose** Set storage space for nonzero elements

**C Syntax**

```
#include "matrix.h"
void mxSetNzmax(mxArray *pm, mwSize nzmax);
```

**Fortran Syntax**

```
subroutine mxSetNzmax(pm, nzmax)
mwPointer pm
mwSize nzmax
```

**Arguments**

pm  
Pointer to a sparse mxArray.

nzmax  
The number of elements that mxCreateSparse should allocate to hold the arrays pointed to by ir, pr, and pi (if it exists). Set nzmax greater than or equal to the number of nonzero elements in the mxArray, but set it to be less than or equal to the number of rows times the number of columns. If you specify an nzmax value of 0, mxSetNzmax sets the value of nzmax to 1.

**Description** Use mxSetNzmax to assign a new value to the nzmax field of the specified sparse mxArray. The nzmax field holds the maximum possible number of nonzero elements in the sparse mxArray.

The number of elements in the ir, pr, and pi (if it exists) arrays must be equal to nzmax. Therefore, after calling mxSetNzmax, you must change the size of the ir, pr, and pi arrays. To change the size of one of these arrays:

- 1 Call mxRealloc with a pointer to the array, setting the size to the new value of nzmax.
- 2 Call the appropriate mxSet routine (mxSetIr, mxSetPr, or mxSetPi) to establish the new memory area as the current one.

Two ways of determining how big you should make nzmax are

- Set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space.
- Make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

### **C Examples**

See `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

### **See Also**

`mxGetNzmax`, `mxRealloc`

# mxSetPi (C and Fortran)

---

<b>Purpose</b>	Set new imaginary data for mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" void mxSetPi(mxArray *pm, double *pi);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxSetPi(pm, pi) mwPointer pm, pi</pre>
<b>Arguments</b>	<p>pm Pointer to a full (nonsparse) mxArray</p> <p>pi Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pi points to static memory, memory leaks and other memory errors may result.</p>
<b>Description</b>	<p>Use mxSetPi to set the imaginary data of the specified mxArray.</p> <p>Most mxCreate functions optionally allocate heap space to hold imaginary data. If you tell an mxCreate function to allocate heap space—for example, by setting the ComplexFlag to mxCOMPLEX in C (1 in Fortran) or by setting pi to a non-NULL value in C (a nonzero value in Fortran)—you do not ordinarily use mxSetPi to initialize the created mxArray's imaginary elements. Rather, you call mxSetPi to replace the initial imaginary values with new ones.</p> <p>This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetPi before you call mxSetPi.</p>
<b>C Examples</b>	See mxisfinite.c and mxsetnzmax.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxGetPi, mxGetPr, mxSetImagData, mxSetPr, mxFree

<b>Purpose</b>	Set new real data for mxArray
<b>C Syntax</b>	<pre>#include "matrix.h" void mxSetPr(mxArray *pm, double *pr);</pre>
<b>Fortran Syntax</b>	<pre>subroutine mxSetPr(pm, pr) mwPointer pm, pr</pre>
<b>Arguments</b>	<p>pm Pointer to a full (nonsparse) mxArray</p> <p>pr Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call mxMalloc to allocate this dynamic memory. If pr points to static memory, memory leaks and other memory errors can result.</p>
<b>Description</b>	<p>Use mxSetPr to set the real data of the specified mxArray.</p> <p>All mxCreate calls allocate heap space to hold real data. Therefore, you do not ordinarily use mxSetPr to initialize the real elements of a freshly created mxArray. Rather, you call mxSetPr to replace the initial real values with new ones.</p> <p>This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetPr before you call mxSetPr.</p>
<b>C Examples</b>	See mxsetnzmax.c in the mx subdirectory of the examples directory.
<b>See Also</b>	mxGetPi, mxGetPr, mxSetData, mxSetPi, mxFree



## A

allocating memory 2-67

## B

buffer

defining output 2-10

## D

deleting

named matrix from MAT-file 2-16

directory

getting 2-17

## E

engClose 2-2

engEvalString 2-3

engGetVariable 2-5

engGetVisible 2-6

engines

getting and putting matrices into 2-5 2-12

starting 2-2

engOpen 2-7

engPutVariable 2-12

engSetVisible 2-14

errors

control response to 2-55

issuing messages 2-34 2-36

## G

getting

directory 2-17

## M

MAT-files

deleting named matrix from 2-16

getting and putting matrices into 2-23  
2-28 to 2-29

getting next matrix from 2-20

getting pointer to 2-19

opening and closing 2-15 2-26

matClose 2-26

matDeleteMatrix 2-16

matGetDir 2-17

matGetFp 2-19

matGetNextVariable 2-20

matGetNextVariableInfo 2-21

matGetVariable 2-23

matGetVariableInfo 2-24

MATLAB engines

starting 2-2

matOpen 2-15

matPutVariable 2-28

matPutVariableAsGlobal 2-29

matrices

putting into engine's workspace 2-12

putting into MAT-files 2-29

MEX-files

entry point to 2-38

mexCallMATLAB 2-32

mexErrMsgIdAndTxt 2-34 2-58

mexErrMsgTxt 2-36 2-59

mexEvalString 2-37

mexFunction 2-38

mexGetVariable 2-42

mexPrintf 2-50

mexSetTrapFlag 2-55

## O

opening MAT-files 2-15 2-26

## P

pointer

to MAT-file 2-19

printing 2-47 2-49

**S**

starting

MATLAB engines 2-2

string

executing statement 2-3