# MATLAB®

**The Language of Technical Computing**

- Computation
- Visualization
- Programming

# External Interfaces Reference

*Version 7*

**How to Contact The MathWorks:**

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

*MATLAB External Interfaces Reference*

**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) |

# Contents

## Functions — By Category

1



## Functions — Alphabetical List

2

## Index

# Functions — By Category



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 (C)

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

## MX Array Manipulation (C)

| | |
|---|---|
| mxAddField (C) | Add field to structure array |
| mxArrayToString (C) | Convert array to string |
| mxAssert (C) | Check assertion value for debugging purposes |
| mxAssertS (C) | Check assertion value without printing assertion text |
| mxCalcSingleSubscript (C) | Offset from first element to desired element |
| mxCalloc (C) | Allocate dynamic memory for array using MATLAB memory manager |
| mxChar (C) | Data type for string mxArray |

| | |
|---|---|
| `mxClassID (C)` | Integer value identifying class of `mxArray` |
| `mxComplexity (C)` | Flag specifying whether `mxArray` has imaginary components |
| `mxCreateCellArray (C)` | Create unpopulated N-D cell `mxArray` |
| `mxCreateCellMatrix (C)` | Create unpopulated 2-D cell `mxArray` |
| `mxCreateCharArray (C)` | Create unpopulated N-D string `mxArray` |
| `mxCreateCharMatrixFromStrings (C)` | Create populated 2-D string `mxArray` |
| `mxCreateDoubleMatrix (C)` | Create unpopulated 2-D, double-precision, floating-point `mxArray` |
| `mxCreateDoubleScalar (C)` | Create scalar, double-precision array initialized to specified value |
| `mxCreateLogicalArray (C)` | Create N-D logical `mxArray` initialized to `false` |
| `mxCreateLogicalMatrix (C)` | Create 2-D, logical `mxArray` initialized to `false` |
| `mxCreateLogicalScalar (C)` | Create scalar, logical `mxArray` initialized to `false` |
| `mxCreateNumericArray (C)` | Create unpopulated N-D numeric `mxArray` |
| `mxCreateNumericMatrix (C)` | Create numeric matrix and initialize data elements to 0 |
| `mxCreateSparse (C)` | Create 2-D unpopulated sparse `mxArray` |
| `mxCreateSparseLogicalMatrix (C)` | Create unpopulated 2-D, sparse, logical `mxArray` |
| `mxCreateString (C)` | Create 1-by-N string `mxArray` initialized to specified string |

| | |
|---|---|
| mxCreateStructArray (C) | Create unpopulated N-D structure mxArray |
| mxCreateStructMatrix (C) | Create unpopulated 2-D structure mxArray |
| mxDestroyArray (C) | Free dynamic memory allocated by mxCreate |
| mxDuplicateArray (C) | Make deep copy of array |
| mxFree (C) | Free dynamic memory allocated by mxCalloc, mxMalloc, or mxRealloc |
| mxGetCell (C) | Contents of mxArray cell |
| mxGetChars (C) | Pointer to character array data |
| mxGetClassID (C) | Class of mxArray |
| mxGetClassName (C) | Class of mxArray as string |
| mxGetData (C) | Pointer to data |
| mxGetDimensions (C) | Pointer to dimensions array |
| mxGetElementSize (C) | Number of bytes required to store each data element |
| mxGetEps (C) | Value of eps |
| mxGetField (C) | Field value, given field name and index into structure array |
| mxGetFieldByNumber (C) | Field value, given field number and index into structure array |
| mxGetFieldNameByNumber (C) | Field name, given field number in structure array |
| mxGetFieldNumber (C) | Field number, given field name in structure array |
| mxGetImagData (C) | Pointer to imaginary data of mxArray |
| mxGetInf (C) | Value of infinity |
| mxGetIr (C) | ir array of sparse matrix |
| mxGetJc (C) | jc array of sparse matrix |

| | |
|---|---|
| `mxGetLogicals (C)` | Pointer to logical array data |
| `mxGetM (C)` | Number of rows in `mxArray` |
| `mxGetN (C)` | Number of columns in `mxArray` |
| `mxGetNaN (C)` | Value of `NaN` (Not-a-Number) |
| `mxGetNumberOfDimensions (C)` | Number of dimensions in `mxArray` |
| `mxGetNumberOfElements (C)` | Number of elements in `mxArray` |
| `mxGetNumberOfFields (C)` | Number of fields in structure `mxArray` |
| `mxGetNzmax (C)` | Number of elements in `ir`, `pr`, and `pi` arrays |
| `mxGetPi (C)` | Imaginary data elements in `mxArray` |
| `mxGetPr (C)` | Real data elements in `mxArray` |
| `mxGetScalar (C)` | Real component of first data element in `mxArray` |
| `mxGetString (C)` | Copy string `mxArray` to C-style string |
| `mxIsCell (C)` | Determine whether input is cell `mxArray` |
| `mxIsChar (C)` | Determine whether input is string `mxArray` |
| `mxIsClass (C)` | Determine whether `mxArray` is member of specified class |
| `mxIsComplex (C)` | Determine whether data is complex |
| `mxIsDouble (C)` | Determine whether `mxArray` represents data as double-precision, floating-point numbers |
| `mxIsEmpty (C)` | Determine whether `mxArray` is empty |
| `mxIsFinite (C)` | Determine whether input is finite |
| `mxIsFromGlobalWS (C)` | Determine whether `mxArray` was copied from MATLAB global workspace |

| | |
|---|---|
| `mxIsInf (C)` | Determine whether input is infinite |
| `mxIsInt16 (C)` | Determine whether `mxArray` represents data as signed 16-bit integers |
| `mxIsInt32 (C)` | Determine whether `mxArray` represents data as signed 32-bit integers |
| `mxIsInt64 (C)` | Determine whether `mxArray` represents data as signed 64-bit integers |
| `mxIsInt8 (C)` | Determine whether `mxArray` represents data as signed 8-bit integers |
| `mxIsLogical (C)` | Determine whether `mxArray` is of class `mxLogical` |
| `mxIsLogicalScalar (C)` | Determine whether scalar `mxArray` is of class `mxLogical` |
| `mxIsLogicalScalarTrue (C)` | Determine whether scalar `mxArray` of class `mxLogical` is `true` |
| `mxIsNaN (C)` | Determine whether input is `NaN` (Not-a-Number) |
| `mxIsNumeric (C)` | Determine whether `mxArray` is numeric |
| `mxIsSingle (C)` | Determine whether `mxArray` represents data as single-precision, floating-point numbers |
| `mxIsSparse (C)` | Determine whether input is sparse `mxArray` |
| `mxIsStruct (C)` | Determine whether input is structure `mxArray` |
| `mxIsUint16 (C)` | Determine whether `mxArray` represents data as unsigned 16-bit integers |

| | |
|---|---|
| `mxIsUint32 (C)` | Determine whether `mxArray` represents data as unsigned 32-bit integers |
| `mxIsUint64 (C)` | Determine whether `mxArray` represents data as unsigned 64-bit integers |
| `mxIsUint8 (C)` | Determine whether `mxArray` represents data as unsigned 8-bit integers |
| `mxMalloc (C)` | Allocate dynamic memory using MATLAB memory manager |
| `mxRealloc (C)` | Reallocate memory |
| `mxRemoveField (C)` | Remove field from structure array |
| `mxSetCell (C)` | Set value of one cell of `mxArray` |
| `mxSetClassName (C)` | Convert structure array to MATLAB object array |
| `mxSetData (C)` | Set pointer to data |
| `mxSetDimensions (C)` | Modify number of dimensions and size of each dimension |
| `mxSetField (C)` | Set structure array field, given field name and index |
| `mxSetFieldByNumber (C)` | Set structure array field, given field number and index |
| `mxSetImagData (C)` | Set imaginary data pointer for `mxArray` |
| `mxSetIr (C)` | Set `ir` array of sparse `mxArray` |
| `mxSetJc (C)` | Set `jc` array of sparse `mxArray` |
| `mxSetM (C)` | Set number of rows in `mxArray` |
| `mxSetN (C)` | Set number of columns in `mxArray` |
| `mxSetNzmax (C)` | Set storage space for nonzero elements |

| | |
|---|---|
| mxSetPi (C) | Set new imaginary data for mxArray |
| mxSetPr (C) | Set new real data for mxArray |

## MEX-Files (C)

| | |
|---|---|
| mexAtExit (C) | Register function to call when MEX-function cleared or MATLAB terminates |
| mexCallMATLAB (C) | Call MATLAB function or user-defined M-file or MEX-file |
| mexErrMsgIdAndTxt (C) | Issue error message with identifier and return to MATLAB prompt |
| mexErrMsgTxt (C) | Issue error message and return to MATLAB prompt |
| mexEvalString (C) | Execute MATLAB command in caller's workspace |
| mexFunction (C) | Entry point to C MEX-file |
| mexFunctionName (C) | Name of current MEX-function |
| mexGet (C) | Value of specified Handle Graphics® property |
| mexGetVariable (C) | Copy of variable from specified workspace |
| mexGetVariablePtr (C) | Read-only pointer to variable from another workspace |
| mexIsGlobal (C) | Determine whether mxArray has global scope |
| mexIsLocked (C) | Determine whether MEX-file is locked |
| mexLock (C) | Prevent MEX-file from being cleared from memory |

| | |
|---|---|
| `mexMakeArrayPersistent (C)` | Make `mxArray` persist after MEX-file completes |
| `mexMakeMemoryPersistent (C)` | Make allocated memory MATLAB persist after MEX-function completes |
| `mexPrintf (C)` | ANSI C `printf`-style output routine |
| `mexPutVariable (C)` | Copy `mxArray` from MEX-function into specified workspace |
| `mexSet (C)` | Set value of specified Handle Graphics property |
| `mexSetTrapFlag (C)` | Control response of `mexCallMATLAB` to errors |
| `mexUnlock (C)` | Allow MEX-file to be cleared from memory |
| `mexWarnMsgIdAndTxt (C)` | Issue warning message with identifier |
| `mexWarnMsgTxt (C)` | Issue warning message |

# MATLAB Engine (C)

| | |
|---|---|
| `engClose (C)` | Quit MATLAB engine session |
| `engEvalString (C)` | Evaluate expression in string |
| `engGetVariable (C)` | Copy variable from MATLAB engine workspace |
| `engGetVisible (C)` | Determine visibility of MATLAB engine session |
| `engOpen (C)` | Start MATLAB engine session |
| `engOpenSingleUse (C)` | Start MATLAB engine session for single, nonshared use |
| `engOutputBuffer (C)` | Specify buffer for MATLAB output |

| | |
|---|---|
| engPutVariable (C) | Put variables into MATLAB engine workspace |
| engSetVisible (C) | Show or hide MATLAB engine session |

## MAT-File Access (Fortran)

| | |
|---|---|
| matClose (Fortran) | Close MAT-file |
| matDeleteVariable (Fortran) | Delete named mxArray from MAT-file |
| matGetDir (Fortran) | Directory of mxArrays from MAT-file |
| matGetNextVariable (Fortran) | Read next mxArray from MAT-file |
| matGetNextVariableInfo (Fortran) | Load array header information only |
| matGetVariable (Fortran) | Read mxArrays from MAT-files |
| matGetVariableInfo (Fortran) | Load array header information only |
| matOpen (Fortran) | Open MAT-file |
| matPutVariable (Fortran) | Write mxArrays to MAT-files |
| matPutVariableAsGlobal (Fortran) | Put mxArrays into MAT-files as originating from global workspace |

## MX Array Manipulation (Fortran)

| | |
|---|---|
| MWPOINTER (Fortran) | Declare appropriate pointer type for platform |
| mxAddField (Fortran) | Add field to structure array |
| mxCalcSingleSubscript (Fortran) | Offset from first element to desired element |

| | |
|---|---|
| mxCalloc (Fortran) | Allocate dynamic memory for array using MATLAB memory manager |
| mxClassIDFromClassName (Fortran) | Identifier corresponding to class |
| mxCopyCharacterToPtr (Fortran) | Copy character values from Fortran array to pointer array |
| mxCopyComplex16ToPtr (Fortran) | Copy COMPLEX*16 values from Fortran array to pointer array |
| mxCopyComplex8ToPtr (Fortran) | Copy COMPLEX*8 values from Fortran array to pointer array |
| mxCopyInteger1ToPtr (Fortran) | Copy INTEGER*1 values from Fortran array to pointer array |
| mxCopyInteger2ToPtr (Fortran) | Copy INTEGER*2 values from Fortran array to pointer array |
| mxCopyInteger4ToPtr (Fortran) | Copy INTEGER*4 values from Fortran array to pointer array |
| mxCopyPtrToCharacter (Fortran) | Copy character values from pointer array to Fortran array |
| mxCopyPtrToComplex16 (Fortran) | Copy COMPLEX*16 values from pointer array to Fortran array |
| mxCopyPtrToComplex8 (Fortran) | Copy COMPLEX*8 values from pointer array to Fortran array |
| mxCopyPtrToInteger1 (Fortran) | Copy INTEGER*1 values from pointer array to Fortran array |
| mxCopyPtrToInteger2 (Fortran) | Copy INTEGER*2 values from pointer array to Fortran array |
| mxCopyPtrToInteger4 (Fortran) | Copy INTEGER*4 values from pointer array to Fortran array |
| mxCopyPtrToPtrArray (Fortran) | Copy pointer values from pointer array to Fortran array |
| mxCopyPtrToReal4 (Fortran) | Copy REAL*4 values from pointer array to Fortran array |

| | |
|---|---|
| mxCopyPtrToReal8 (Fortran) | Copy REAL*8 values from pointer array to Fortran array |
| mxCopyReal4ToPtr (Fortran) | Copy REAL*4 values from Fortran array to pointer array |
| mxCopyReal8ToPtr (Fortran) | Copy REAL*8 values from Fortran array to pointer array |
| mxCreateCellArray (Fortran) | Create unpopulated N-D cell mxArray |
| mxCreateCellMatrix (Fortran) | Create unpopulated 2-D cell mxArray |
| mxCreateCharArray (Fortran) | Create unpopulated N-D character mxArray |
| mxCreateCharMatrixFromStrings (Fortran) | Create populated 2-D char mxArray |
| mxCreateDoubleMatrix (Fortran) | Create unpopulated 2-D, double-precision, floating-point mxArray |
| mxCreateDoubleScalar (Fortran) | Create scalar, double-precision array initialized to specified value |
| mxCreateNumericArray (Fortran) | Create unpopulated N-D numeric mxArray |
| mxCreateNumericMatrix (Fortran) | Create numeric matrix and initialize data elements to 0 |
| mxCreateSparse (Fortran) | Create 2-D unpopulated sparse mxArray |
| mxCreateString (Fortran) | Create 1-by-N character array initialized to specified string |
| mxCreateStructArray (Fortran) | Create unpopulated N-D structure mxArray |
| mxCreateStructMatrix (Fortran) | Create unpopulated 2-D structure mxArray |
| mxDestroyArray (Fortran) | Free dynamic memory allocated by mxCreate |

| | |
|---|---|
| mxDuplicateArray (Fortran) | Make deep copy of array |
| mxFree (Fortran) | Free dynamic memory allocated by mxCalloc, mxMalloc, or mxRealloc |
| mxGetCell (Fortran) | Contents of cell |
| mxGetClassID (Fortran) | Class identifier of mxArray |
| mxGetClassName (Fortran) | mxArray class as character array |
| mxGetData (Fortran) | Pointer to data |
| mxGetDimensions (Fortran) | Pointer to dimensions array |
| mxGetElementSize (Fortran) | Number of bytes required to store each data element |
| mxGetEps (Fortran) | Value of eps |
| mxGetField (Fortran) | Structure array field value, given field name and index |
| mxGetFieldByNumber (Fortran) | Structure array field value, given field number and index |
| mxGetFieldNameByNumber (Fortran) | Structure array field name, given field number |
| mxGetFieldNumber (Fortran) | Structure array field number, given field name |
| mxGetImagData (Fortran) | Pointer to imaginary data of mxArray |
| mxGetInf (Fortran) | Value of infinity |
| mxGetIr (Fortran) | ir array |
| mxGetJc (Fortran) | jc array |
| mxGetM (Fortran) | Number of rows in mxArray |
| mxGetN (Fortran) | Number of columns in mxArray |
| mxGetNaN (Fortran) | Value of NaN (Not-a-Number) |
| mxGetNumberOfDimensions (Fortran) | Number of dimensions in mxArray |
| mxGetNumberOfElements (Fortran) | Number of elements in mxArray |

| | |
|---|---|
| mxGetNumberOfFields (Fortran) | Number of fields in structure mxArray |
| mxGetNzmax (Fortran) | Number of elements in ir, pr, and pi arrays |
| mxGetPi (Fortran) | Imaginary data elements of mxArray |
| mxGetPr (Fortran) | Real data elements of mxArray |
| mxGetScalar (Fortran) | Real component of first data element in mxArray |
| mxGetString (Fortran) | Create character array from mxArray |
| mxIsCell (Fortran) | Determine whether input is cell mxArray |
| mxIsChar (Fortran) | Determine whether input is character mxArray |
| mxIsClass (Fortran) | Determine whether mxArray is member of specified class |
| mxIsComplex (Fortran) | Determine whether mxArray is complex |
| mxIsDouble (Fortran) | Determine whether mxArray is of type double |
| mxIsEmpty (Fortran) | Determine whether mxArray is empty |
| mxIsFinite (Fortran) | Determine whether input is finite |
| mxIsFromGlobalWS (Fortran) | Determine whether mxArray originated from MATLAB global workspace |
| mxIsInf (Fortran) | Determine whether input is infinite |
| mxIsInt16 (Fortran) | Determine whether input is mxArray of signed 16-bit integers |
| mxIsInt32 (Fortran) | Determine whether input is mxArray of signed 32-bit integers |

| | |
|---|---|
| `mxIsInt64 (Fortran)` | Determine whether input is `mxArray` of signed 64-bit integers |
| `mxIsInt8 (Fortran)` | Determine whether input is `mxArray` of signed 8-bit integers |
| `mxIsLogical (Fortran)` | Determine whether `mxArray` is Boolean |
| `mxIsNaN (Fortran)` | Determine whether value is `NaN` (Not-a-Number) |
| `mxIsNumeric (Fortran)` | Determine whether `mxArray` contains numeric data |
| `mxIsSingle (Fortran)` | Determine whether input is single-precision, floating-point `mxArray` |
| `mxIsSparse (Fortran)` | Determine whether `mxArray` is sparse |
| `mxIsStruct (Fortran)` | Determine whether input is structure `mxArray` |
| `mxIsUint16 (Fortran)` | Determine whether input is `mxArray` of unsigned 16-bit integers |
| `mxIsUint32 (Fortran)` | Determine whether input is `mxArray` of unsigned 32-bit integers |
| `mxIsUint64 (Fortran)` | Determine whether input is `mxArray` of unsigned 64-bit integers |
| `mxIsUint8 (Fortran)` | Determine whether input is `mxArray` of unsigned 8-bit integers |
| `mxMalloc (Fortran)` | Allocate dynamic memory using MATLAB memory manager |
| `mxRealloc (Fortran)` | Reallocate memory |
| `mxRemoveField (Fortran)` | Remove field from structure `mxArray` |
| `mxSetCell (Fortran)` | Set value of one cell of cell `mxArray` |
| `mxSetData (Fortran)` | Set pointer to data |

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

## MEX-Files (Fortran)

| | |
|---|---|
| mexAtExit (Fortran) | Register subroutine to call when MEX-file cleared or MATLAB terminates |
| mexCallMATLAB (Fortran) | Call MATLAB function or operator, user-defined M-file, or other MEX-file |
| mexErrMsgIdAndTxt (Fortran) | Issue error with identifier and return to MATLAB prompt |
| mexErrMsgTxt (Fortran) | Issue error and return to MATLAB prompt |
| mexEvalString (Fortran) | Execute MATLAB command in caller's workspace |

| | |
|---|---|
| `mexFunction (Fortran)` | MATLAB entry point to Fortran MEX-file |
| `mexFunctionName (Fortran)` | Name of current MEX-function |
| `mexGetVariable (Fortran)` | Copy of variable from specified workspace |
| `mexGetVariablePtr (Fortran)` | Read-only pointer to variable from specified workspace |
| `mexIsGlobal (Fortran)` | Determine whether `mxArray` has global scope |
| `mexIsLocked (Fortran)` | Determine whether MEX-file is locked |
| `mexLock (Fortran)` | Prevent MEX-file from being cleared from memory |
| `mexMakeArrayPersistent (Fortran)` | Make `mxArray` persist after MEX-file completes |
| `mexMakeMemoryPersistent (Fortran)` | Make allocated memory persist after MEX-file completes |
| `mexPrintf (Fortran)` | Print `character` array |
| `mexPutVariable (Fortran)` | Copy `mxArray` into specified workspace |
| `mexSetTrapFlag (Fortran)` | Control response of `mexCallMATLAB` to errors |
| `mexUnlock (Fortran)` | Allow MEX-file to be cleared from memory |
| `mexWarnMsgIdAndTxt (Fortran)` | Issue warning message with identifier |
| `mexWarnMsgTxt (Fortran)` | Issue warning message |

# MATLAB Engine (Fortran)

| | |
|---|---|
| `engClose (Fortran)` | Quit MATLAB engine session |
| `engEvalString (Fortran)` | Evaluate expression in `character` array |
| `engGetVariable (Fortran)` | Copy variable from MATLAB engine workspace |
| `engOpen (Fortran)` | Start MATLAB engine session |
| `engOutputBuffer (Fortran)` | Specify buffer for MATLAB output |
| `engPutVariable (Fortran)` | Put variables into MATLAB engine workspace |

2

# Functions — Alphabetical List

# engClose (C)

## Purpose

Quit MATLAB engine session

## C Syntax

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

## 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.

## Purpose

Quit MATLAB engine session

## 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.

## 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.

# engEvalString (C)

## Purpose

Evaluate expression in string

## C Syntax

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

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

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

## 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.

## Purpose

Evaluate expression in `character` array

## Fortran Syntax

```
integer*4 function engEvalString(ep, command)
MWPOINTER ep
character*(*) command
```

## Arguments

`ep`
: Engine pointer.

`command`
: `character` array to execute.

## Description

`engEvalString` evaluates the expression contained in `command` 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`.

## 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 (Fortran)`, `engOutputBuffer (Fortran)`

# engGetVariable (C)

## Purpose

Copy variable from MATLAB engine workspace

## C Syntax

```
#include "engine.h"
mxArray *engGetVariable(Engine *ep, const char *var_name);
```

## Arguments

`ep`
Engine pointer.

`var_name`
Name of `mxArray` to get from MATLAB.

## Description

`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.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

## 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

`engPutVariable (C)`

## Purpose

Copy variable from MATLAB engine workspace

## Fortran Syntax

```
MWPOINTER function engGetVariable(ep, name)
MWPOINTER ep
character*(*) name
```

## Arguments

`ep`
: Engine pointer.

`name`
: Name of `mxArray` to get from MATLAB.

## Description

`engGetVariable` reads the named `mxArray` from the MATLAB engine session associated with `ep` and returns a pointer to a newly allocated `mxArray` structure, or `0` if the attempt fails. `engGetVariable` fails if the named variable does not exist.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

## See Also

`engPutVariable (Fortran)`

# 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 (C)`

## Purpose

Start MATLAB engine session

## C Syntax

```
#include "engine.h"
Engine *engOpen(const char *startcmd);
```

## Arguments

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

## Returns

A pointer to an engine handle.

## Description

This routine allows you to start a MATLAB process for the purpose of using MATLAB as a computational engine.

`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.

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:

```
"rsh hostname \"/bin/csh -c 'setenv DISPLAY\
 hostname:0; matlab'\""
```

If `startcmd` is any other string (has white space in it, or nonalphanumeric characters), the string is executed literally to start MATLAB.

On UNIX systems, `engOpen` performs the following steps:

1. Creates two pipes.
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.

## 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.

## Purpose

Start MATLAB engine session

## Fortran Syntax

```
MWPOINTER function engOpen(startcmd)
character*(*) startcmd
```

## Arguments

`startcmd`
: `Character` array to start a MATLAB process.

## Description

This routine allows you to start a MATLAB process to use MATLAB as a computational engine.

`engOpen(startcmd)` starts a MATLAB process using the command specified in `startcmd`, establishes a connection, and returns a unique engine identifier, or 0 if the open fails.

On the UNIX system, if `startcmd` is empty, `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:

```
"rsh hostname \"/bin/csh -c 'setenv DISPLAY\
 hostname:0; matlab'\""
```

If `startcmd` is anything else (has white space in it, or nonalphanumeric characters), it is executed literally to start MATLAB.

`engOpen` performs the following steps:

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

## 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.

# engOpenSingleUse (C)

## 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.

## Purpose

Specify buffer for MATLAB output

## C Syntax

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

## Arguments

`ep`
: Engine pointer.

`p`
: Pointer to character buffer of length `n`.

`n`
: Length of buffer `p`.

## Description

`engOutputBuffer` defines a character buffer for `engEvalString` to return any output that ordinarily appears on the screen.

The default behavior of `engEvalString` is to discard any standard output caused by the command it is executing. `engOutputBuffer(ep, p, n)` tells any subsequent calls to `engEvalString` to save the first `n` characters of output in the character buffer pointed to by `p`.

To turn off output buffering, use `engOutputBuffer(ep, NULL, 0);`

---

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

---

## 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.

# engOutputBuffer (Fortran)

## Purpose

Specify buffer for MATLAB output

## Fortran Syntax

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

## Arguments

`ep`

Engine pointer.

`p`

Character buffer of length `n`, where `n` is the length of buffer `p`.

## Description

`engOutputBuffer` defines a character buffer for `engEvalString` to return any output that would appear 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. `engOutputBuffer(ep, p)` tells any subsequent calls to `engEvalString` to save the first `n` characters of output in the character buffer `p`.

## Purpose

Put variables into MATLAB engine workspace

## C Syntax

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

## Arguments

`ep`
Engine pointer.

`var_name`
Name given to the `mxArray` in the engine's workspace.

`array_ptr`
`mxArray` pointer.

## Description

`engPutVariable` writes `mxArray` `array_ptr` to the engine `ep`, giving it the variable name `var_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.

## 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.

# engPutVariable (Fortran)

## Purpose

Put variables into MATLAB engine workspace

## 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`. 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.

## See Also

`engGetVariable (Fortran)`

## Purpose

Show or hide MATLAB engine session

## C Syntax

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

## Arguments

`ep`
: Engine pointer.

`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.

## Description

### Windows Only

`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.

`engSetVisible` 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

`engGetVisible (C)`

# matClose (C)

## Purpose

Close MAT-file

## C Syntax

```
#include "mat.h"
int matClose(MATFile *mfp);
```

## Arguments

`mfp`
Pointer to MAT-file information.

## Description

`matClose` closes the MAT-file associated with `mfp`. It returns `EOF` for a write error, and zero if successful.

## 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.

## Purpose

Close MAT-file

## Fortran Syntax

```
integer*4 function matClose(mfp)
MWPOINTER mfp
```

## Arguments

`mfp`
Pointer to MAT-file information.

## Description

`matClose` closes the MAT-file associated with `mfp`. It returns -1 for a write error, and 0 if successful.

## Examples

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)

## Purpose

Delete named mxArray from MAT-file

## C Syntax

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

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of mxArray to delete.

## Description

`matDeleteVariable` deletes the named `mxArray` from the MAT-file pointed to by `mfp`.`matDeleteVariable` returns 0 if successful, and nonzero otherwise.

## 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.

## Purpose

Delete named mxArray from MAT-file

## Fortran Syntax

```
integer*4 function matDeleteVariable(mfp, name)
MWPOINTER mfp
character*(*) name
```

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of `mxArray` to delete.

## Description

`matDeleteVariable` deletes the named `mxArray` from the MAT-file pointed to by `mfp`. The function returns `0` if successful, and nonzero otherwise.

# matGetDir (C)

## Purpose

Directory of mxArrays in MAT-file

## C Syntax

```
#include "mat.h"
char **matGetDir(MATFile *mfp, int *num);
```

## Arguments

mfp
: Pointer to MAT-file information.

num
: Address of the variable to contain the number of mxArrays in the MAT-file.

## Description

This routine allows you to get a list of the names of the mxArrays contained within a MAT-file.

matGetDir returns a pointer to an internal array containing pointers to the NULL-terminated names of the mxArrays in the MAT-file pointed to by mfp. The length of the internal array (number of mxArrays in the MAT-file) is placed into num. The internal array is allocated using a single mxCalloc and must be freed using mxFree when you are finished with it.

matGetDir returns NULL and sets num to a negative number if it fails. If num is zero, mfp contains no arrays.

MATLAB variable names can be up to length mxMAXNAM, where mxMAXNAM is defined in the file matrix.h.

## 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.

## Purpose

Directory of mxArrays from MAT-file

## Fortran Syntax

```
MWPOINTER function matGetDir(mfp, num)
MWPOINTER mfp
integer*4 num
```

## Arguments

`mfp`
: Pointer to MAT-file information.

`num`
: Address of the variable to contain the number of mxArrays in the MAT-file.

## Description

This routine enables you to get a list of the names of the mxArrays contained within a MAT-file.

matGetDir returns a pointer to an internal array containing pointers to the names of the mxArrays in the MAT-file pointed to by mfp. The length of the internal array (number of mxArrays in the MAT-file) is placed into num. The internal array is allocated using a single mxCalloc. Use mxFree to free the array when you are finished with it.

matGetDir returns 0 and sets num to a negative number if it fails. If num is zero, mfp contains no mxArrays.

MATLAB variable names can be up to length 32.

## 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.

# matGetFp (C)

## Purpose

File pointer to MAT-file

## C Syntax

```
#include "mat.h"
FILE *matGetFp(MATFile *mfp);
```

## Arguments

`mfp`
Pointer to MAT-file information.

## Description

`matGetFp` returns the C file handle to the MAT-file with handle `mfp`. This can be useful for using standard C library routines like `ferror()` and `feof()` to investigate error situations.

## 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.

## Purpose

Read next mxArray from MAT-file

## C Syntax

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

## Arguments

mfp
: Pointer to MAT-file information.

name
: Address of the variable to contain the mxArray name.

## Description

matGetNextVariable allows you to step sequentially through a MAT-file and read all the mxArrays in a single pass. The function reads the next mxArray from the MAT-file pointed to by mfp and returns a pointer to a newly allocated mxArray structure. MATLAB returns the name of the mxArray in name.

Use matGetNextVariable immediately after opening the MAT-file with matOpen and not in conjunction with other MAT-file routines. Otherwise, the concept of the *next* mxArray is undefined.

matGetNextVariable returns NULL when the end-of-file is reached or if there is an error condition. Use feof and ferror from the Standard C Library to determine status.

Be careful in your code to free the mxArray created by this routine when you are finished with it.

## 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.

# matGetNextVariable (Fortran)

## Purpose

Read next mxArray from MAT-file

## Fortran Syntax

```
MWPOINTER function matGetNextVariable(mfp, name)
MWPOINTER mfp
character*(*) name
```

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Address of the variable to contain the mxArray name.

## Description

matGetNextVariable allows you to step sequentially through a MAT-file and read all the mxArrays in a single pass. The function reads the next mxArray from the MAT-file pointed to by mfp and returns a pointer to a newly allocated mxArray structure. MATLAB returns the name of the mxArray in name.

Use matGetNextVariable immediately after opening the MAT-file with matOpen and not in conjunction with other MAT-file routines. Otherwise, the concept of the *next* mxArray is undefined.

matGetNextVariable returns 0 when the end-of-file is reached or if there is an error condition.

Be careful in your code to free the mxArray created by this routine when you are finished with it.

## Purpose

Load array header information only

## C Syntax

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

## Arguments

`mfp`
Pointer to MAT-file information.

`name`
Address of the variable to contain the `mxArray` name.

## Description

`matGetNextVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`, from the file's current file offset. MATLAB returns the name of the `mxArray` in `name`.

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 *never* be passed back to MATLAB or saved to MAT-files.

## 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.

## See Also

`matGetNextVariable (C)`, `matGetVariableInfo (C)`

# matGetNextVariableInfo (Fortran)

## Purpose

Load array header information only

## Fortran Syntax

```
MWPOINTER function matGetNextVariableInfo(mfp, name)
MWPOINTER mfp
character*(*) name
```

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Address of the variable to contain the `mxArray` name.

## Description

`matGetNextVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`, from the file's current file offset. MATLAB returns the name of the `mxArray` in `name`.

If `pr`, `pi`, `ir`, and `jc` are set to nonzero values when loaded with `matGetVariable (Fortran)`, `matGetNextVariableInfo` sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

## Purpose

Read mxArrays from MAT-files

## C Syntax

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

## Arguments

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray` to get from MAT-file.

## Description

This routine allows you to copy an `mxArray` out of a MAT-file.

`matGetVariable` reads the named `mxArray` from the MAT-file pointed to by `mfp` and returns a pointer to a newly allocated `mxArray` structure, or `NULL` if the attempt fails.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

## 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.

# matGetVariable (Fortran)

## Purpose

Read mxArrays from MAT-files

## Fortran Syntax

```
MWPOINTER function matGetVariable(mfp, name)
MWPOINTER mfp
character*(*) name
```

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of `mxArray` to get from MAT-file.

## Description

This routine allows you to copy an `mxArray` out of a MAT-file.

`matGetVariable` reads the named `mxArray` from the MAT-file pointed to by `mfp` and returns a pointer to a newly allocated `mxArray` structure, or `0` if the attempt fails.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

## Purpose

Load array header information only

## C Syntax

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

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of `mxArray`.

## Description

`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`.

If `pr`, `pi`, `ir`, and `jc` are set to non-`NULL` when loaded with `matGetVariable`, then `matGetVariableInfo` sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

## 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.

## See Also

`matGetVariable (C)`

# matGetVariableInfo (Fortran)

## Purpose

Load array header information only

## Fortran Syntax

```
MWPOINTER function matGetVariableInfo(mfp, name);
MWPOINTER mfp
character*(*) name
```

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of `mxArray`.

## Description

`matGetVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`. It recursively creates the cells/structures through their leaf elements, but does not include `pr`, `pi`, `ir`, and `jc`.

If `pr`, `pi`, `ir`, and `jc` are set to nonzero values when loaded with `matGetVariable (Fortran)`, `matGetVariableInfo` sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

## Purpose

Open MAT-file

## C Syntax

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

## Arguments

`filename`
Name of file to open.

`mode`
File opening mode. Valid values for `mode` are

| | |
|---|---|
| `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.<br>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. |

## Description

This routine allows you to open MAT-files for reading and writing.

`matOpen` opens the named file and returns a file handle, or `NULL` if the open fails.

See "Writing Character Data" in the External Interfaces documentation for more information on how MATLAB uses character encodings.

## 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.

## Purpose

Open MAT-file

## Fortran Syntax

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

## Arguments

`filename`

Name of file to open.

`mode`

File opening mode. Legal values for `mode` are

| | |
|---|---|
| `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.<br>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. |

# matOpen (Fortran)

## Description

This routine allows you to open MAT-files for reading and writing.

`matOpen` opens the named file and returns a file handle, or 0 if the open fails.

## 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.

## Purpose

Write mxArrays to MAT-files

## C Syntax

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

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of `mxArray` to put into MAT-file.

`mp`
: `mxArray` pointer.

## Description

This routine allows you to put an `mxArray` into a MAT-file.

`matPutVariable` writes `mxArray mp` 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`.

`matPutVariable` returns `0` if successful and nonzero if an error occurs. Use `feof` and `ferror` from the Standard C Library along with `matGetFp` to determine status.

## 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.

# matPutVariable (Fortran)

## Purpose

Write mxArrays to MAT-files

## 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.

## 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`.

`matPutVariable` returns `0` if successful and nonzero if an error occurs.

## Purpose

Put mxArrays into MAT-files as originating from global workspace

## C Syntax

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

## Arguments

`mfp`
: Pointer to MAT-file information.

`name`
: Name of `mxArray` to put into MAT-file.

`mp`
: `mxArray` pointer.

## Description

This routine allows you to put an `mxArray` into a MAT-file. `matPutVariableAsGlobal` is similar to `matPutVariable`, except 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` will not load it as global, and will act the same as `matPutVariable`.

`matPutVariableAsGlobal` writes `mxArray mp` 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`.

`matPutVariableAsGlobal` returns `0` if successful and nonzero if an error occurs. Use `feof` and `ferror` from the Standard C Library with `matGetFp` to determine status.

## 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.

# matPutVariableAsGlobal (Fortran)

## Purpose

Put mxArrays into MAT-files as originating from global workspace

## Fortran Syntax

```
integer*4 function matPutVariableAsGlobal(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.

## Description

This routine allows you to put 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` will not load it as global, and will act the same as `matPutVariable`.

`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`.

`matPutVariableAsGlobal` returns `0` if successful and nonzero if an error occurs.

## Purpose

Register function to call when MEX-function cleared or MATLAB terminates

## C Syntax

```
#include "mex.h"
int mexAtExit(void (*ExitFcn)(void));
```

## Arguments

`ExitFcn`
Pointer to function you want to run on exit.

## Returns

Always returns 0.

## Description

Use `mexAtExit` to register a C 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.

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.

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`.

## Examples

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

## See Also

`mexLock (C)`, `mexUnlock (C)`

# mexAtExit (Fortran)

## Purpose

Register subroutine to call when MEX-file cleared or MATLAB terminates

## Fortran Syntax

```
integer*4 function mexAtExit(ExitFcn)
subroutine ExitFcn()
```

## Arguments

`ExitFcn`

The exit function. This function must be declared as `external`.

## Returns

Always returns 0.

## Description

Use `mexAtExit` to register a subroutine to be called just before the MEX-file is cleared or MATLAB is terminated. `mexAtExit` gives your MEX-file a chance to perform an orderly shutdown of anything under its control.

Each MEX-file can register only one active exit subroutine at a time. If you call `mexAtExit` more than once, MATLAB uses the `ExitFcn` from the more recent `mexAtExit` call as the exit function.

If a MEX-file 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`.

You must declare the `ExitFcn` as `external` in the Fortran routine that calls `mexAtExit` if it is not within the scope of the file.

## See Also

`mexSetTrapFlag (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 *command_name);
```

## Arguments

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

`plhs`
: Pointer to an array of `mxArrays`. The called command puts pointers to the resultant `mxArrays` into `plhs`. Note that the called command 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`
: Pointer to an array of input arguments.

`command_name`
: Character string containing the name of the MATLAB built-in, operator, M-file, or MEX-file that you are calling. If `command_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 command_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.

Note that 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 only assigns 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.

## Examples

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

For additional examples, see sincall.c in the refbook subdirectory of the examples directory; see mexevalstring.c and mexsettrapflag.c in the mex subdirectory of the examples directory; see mxcreatecellmatrix.c and mxisclass.c in the mx subdirectory of the examples directory.

## See Also

mexFunction (C), mexSetTrapFlag (C)

## Purpose

Call MATLAB function or operator, user-defined M-file, or other MEX-file

## 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 `mxArray` pointers that can be used to access the returned data from the function call. Once the data is accessed, you can then call `mxFree` to free the `mxArray` pointer. By default, MATLAB frees the pointer and any associated dynamic memory it allocates when you return from the `mexFunction` call.

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

`prhs`
: Array of pointers to input data.

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

## Returns

0 if successful, and a nonzero value if unsuccessful and `mexSetTrapFlag` was previously called.

## Description

Call `mexCallMATLAB` to invoke internal MATLAB functions, MATLAB operators, M-files, or other MEX-files.

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`.

## See Also

`mexFunction (Fortran)`, `mexSetTrapFlag (Fortran)`

## Purpose

Issue error message with identifier and return to MATLAB prompt

## C Syntax

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

## Arguments

`identifier`
: String containing a MATLAB message identifier. See Message Identifiers in the MATLAB documentation for information on this topic.

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

...
: Any additional arguments needed to translate formatting conversion characters used in `error_msg`. Each conversion character in `error_msg` 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 (C)`, `mexWarnMsgIdAndTxt (C)`, `mexWarnMsgTxt (C)`

## Purpose

Issue error with identifier and return to MATLAB prompt

## Fortran Syntax

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

## Arguments

`errorid`

Character array containing a MATLAB message identifier. See Message Identifiers in the MATLAB documentation for information on this topic.

`errormsg`

`Character` array containing the error message to be displayed.

## 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 any registered exit routine to allocate memory.

If your application calls `mxCalloc` or one of the `mxCreate` routines to create `mxArray` pointers, `mexErrMsgIdAndTxt` automatically frees any associated memory allocated by these calls.

## See Also

`mexErrMsgTxt (Fortran)`, `mexWarnMsgIdAndTxt (Fortran)`, `mexWarnMsgTxt (Fortran)`

# mexErrMsgTxt (C)

## Purpose

Issue error message and return to MATLAB prompt

## C Syntax

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

## Arguments

`error_msg`
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 `mxCreate` 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".

---

## 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 (C)`, `mexWarnMsgTxt (C)`, `mexWarnMsgIdAndTxt (C)`

## Purpose

Issue error and return to MATLAB prompt

## Fortran Syntax

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

## Arguments

`errormsg`

`Character` array 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 any registered exit routine to allocate memory.

If your application calls `mxCalloc` or one of the `mxCreate` routines to create `mxArray` pointers, `mexErrMsgTxt` automatically frees any associated memory allocated by these calls.

## See Also

`mexErrMsgIdAndTxt (Fortran)`, `mexWarnMsgTxt (Fortran)`, `mexWarnMsgIdAndTxt (Fortran)`

# mexEvalString (C)

## Purpose

Execute MATLAB command in caller's workspace

## C Syntax

```
#include "mex.h"
int mexEvalString(const char *command);
```

## Arguments

`command`
A string containing the MATLAB command to execute.

## Returns

0 if successful, and a nonzero value if unsuccessful.

## Description

Call `mexEvalString` to invoke a MATLAB `command` in the workspace of the caller.

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. However, `mexCallMATLAB` provides a mechanism for returning results (left-hand side arguments) back to the MEX-file; `mexEvalString` provides no way for return values to be passed back to the MEX-file.

All arguments that appear to the right of an equals sign in the `command` string must already be current variables of the caller's workspace.

## Examples

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

## See Also

`mexCallMATLAB (C)`

## Purpose

Execute MATLAB command in caller's workspace

## Fortran Syntax

```
integer*4 function mexEvalString(command)
character*(*) command
```

## Arguments

`command`

A `character` array containing the MATLAB command to execute.

## Returns

0 if successful, and a nonzero value if unsuccessful.

## Description

Call `mexEvalString` to invoke a MATLAB command in the workspace of the caller.

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. However, `mexCallMATLAB` provides a mechanism for returning results (left-hand side arguments) back to the MEX-file; `mexEvalString` provides no way for return values to be passed back to the MEX-file.

All arguments that appear to the right of an equals sign in the command array must already be current variables of the caller's workspace.

## See Also

`mexCallMATLAB (Fortran)`

# mexFunction (C)

## Purpose

Entry point to C MEX-file

## C Syntax

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

## Arguments

`nlhs`
: MATLAB sets `nlhs` with the number of expected `mxArrays`.

`plhs`
: MATLAB sets `plhs` to a pointer to an array of NULL pointers.

`nrhs`
: MATLAB sets `nrhs` to the number of input `mxArrays`.

`prhs`
: MATLAB sets `prhs` to a pointer to an array of input `mxArrays`. `These mxArrays` are declared as `constant`; they are read only and should not be modified by your MEX-file. Changing the data in these `mxArrays` may produce undesired side effects.

## Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the generic name of the function entry point that must exist in every C source 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.

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

```
[a,b,c,...] = fun(d,e,f,...)
```

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 a pointer to a length `nrhs` array of pointers to the right-hand side `mxArrays`. `plhs` is a pointer to a length `nlhs` array where your C function must put pointers for the returned left-hand side `mxArrays`.

## Examples

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

# mexFunction (Fortran)

## Purpose

MATLAB entry point to Fortran MEX-file

## Fortran Syntax

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer*4 nlhs, nrhs
MWPOINTER plhs(*), prhs(*)
```

## Arguments

`nlhs`
: The number of expected outputs.

`plhs`
: Array of pointers to expected outputs.

`nrhs`
: The number of inputs.

`prhs`
: Array of pointers to input data. The input data is read only and should not be altered by your `mexFunction`.

## Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of a subroutine you must write in every MEX-file. When you invoke a MEX-file, MATLAB searches for a subroutine named `mexFunction` inside the MEX-file. If it finds one, then the first executable line in `mexFunction` becomes the starting point of the MEX-file. If MATLAB cannot find a subroutine named `mexFunction` inside the MEX-file, MATLAB issues an error message.

When you invoke a MEX-file, MATLAB automatically loads `nlhs`, `plhs`, `nrhs`, and `prhs` with the caller's information. In the syntax of the MATLAB language, functions have the general form

```
[a,b,c,...] = fun(d,e,f,...)
```

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-file is called. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is a pointer to an array whose length is `nlhs`,

where your function must set pointers for the returned left-hand side `mxArrays`.

# mexFunctionName (C)

## Purpose

Name of current MEX-function

## C Syntax

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

## Arguments

none

## Returns

The name of the current MEX-function.

## Description

`mexFunctionName` returns the name of the current MEX-function.

## Examples

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

## Purpose

Name of current MEX-function

## Fortran Syntax

```
character*(*) function mexFunctionName()
```

## Arguments

None

## Returns

The name of the current MEX-function.

## Description

`mexFunctionName` returns the name of the current MEX-function.

# mexGet (C)

## Purpose

Value of specified Handle Graphics® property

## C Syntax

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

## Arguments

`handle`
Handle to a particular graphics object.

`property`
A Handle Graphics property.

## Returns

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 `mxArrays` may produce undesired side effects.

## Description

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`.

## Examples

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

## See Also

`mexSet (C)`

## Purpose

Copy of variable from specified workspace

## C Syntax

```
#include "mex.h"
mxArray *mexGetVariable(const char *workspace, const char
*var_name);
```

## Arguments

`workspace`
Specifies where `mexGetVariable` should search in order to find array, `var_name`. The possible values are

| | |
|---|---|
| `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 |

`var_name`
Name of the variable to copy.

## Returns

A copy of the variable on success. Returns `NULL` 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.

## Description

Call `mexGetVariable` to get a copy of the specified variable. The returned `mxArray` contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned `mxArray` do not affect the variable in the workspace unless you write the copy back to the workspace with `mexPutVariable`.

## Examples

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

## See Also

`mexGetVariablePtr (C)`, `mexPutVariable (C)`

# mexGetVariable (Fortran)

## Purpose

Copy of variable from specified workspace

## Fortran Syntax

```
MWPOINTER function mexGetVariable(workspace, varname)
character*(*) workspace, varname
```

## Arguments

`workspace`

Specifies where `mexGetVariable` should search in order to find variable `varname`. The possible values are

| | |
|---|---|
| `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 |

`varname`

Name of the variable to copy.

## Returns

A copy of the variable on success. Returns `0` on failure. A common cause of failure is specifying a variable that is not currently in the workspace.

## Description

Call `mexGetVariable` to get a copy of the specified variable. The returned `mxArray` contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned `mxArray` do not affect the variable in the workspace unless you write the copy back to the workspace with `mexPutVariable (Fortran)`.

## See Also

`mexGetVariablePtr (Fortran)`, `mexPutVariable (Fortran)`

## Purpose

Read-only pointer to variable from another workspace

## C Syntax

```
#include "mex.h"
const mxArray *mexGetVariablePtr(const char *workspace,
const char *var_name);
```

## Arguments

`workspace`

Specifies which workspace you want `mexGetVariablePtr` to search. The possible values are

| | |
|---|---|
| `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 |

`var_name`

Name of a variable in another workspace. (Note that this is a variable name, not an `mxArray` pointer.)

## Returns

A read-only pointer to the `mxArray` on success. Returns `NULL` on failure.

## Description

Call `mexGetVariablePtr` to get a read-only pointer to the specified variable, `var_name`, 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 (C)` (along with `mexPutVariable (C)`) instead of `mexGetVariablePtr`.

If you simply need to examine data or characteristics, `mexGetVariablePtr` offers superior performance as the caller need pass only a pointer to the array.

## Examples

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

## See Also

`mexGetVariable (C)`

# mexGetVariablePtr (Fortran)

## Purpose

Read-only pointer to variable from specified workspace

## Fortran Syntax

```
MWPOINTER function mexGetVariablePtr(workspace, varname)
character*(*) workspace, varname
```

## Arguments

`workspace`

Specifies which workspace you want `mexGetVariablePtr` to search. The possible values are

| | |
|---|---|
| `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 |

`varname`

Name of the variable to copy. (Note that this is a variable name, not an `mxArray` pointer.)

## Returns

A read-only pointer to the `mxArray` on success. Returns `0` on failure.

## Description

Call `mexGetVariablePtr` to get a read-only pointer to the specified variable `varname` from the specified workspace. This command is useful for examining an `mxArray`'s data and characteristics. If you need to change data or characteristics, use `mexGetVariable (Fortran)` (along with `mexPutVariable (Fortran)`) instead of `mexGetVariablePtr`.

## See Also

`mexGetVariable (Fortran)`

## Purpose

Determine whether mxArray has global scope

## C Syntax

```
#include "matrix.h"
bool mexIsGlobal(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

Logical 1 (true) if the mxArray has global scope, and logical 0 (false) otherwise.

## Description

Use mexIsGlobal to determine if the specified mxArray has global scope.

## Examples

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

## See Also

mexGetVariable (C), mexGetVariablePtr (C), mexPutVariable (C), global

# mexIsGlobal (Fortran)

## Purpose

Determine whether mxArray has global scope

## Fortran Syntax

```
integer*4 function mexIsGlobal(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

Logical 1 (true) if the mxArray has global scope, and logical 0 (false) otherwise.

## Description

Use mexIsGlobal to determine if the specified mxArray has global scope.

## See Also

mexGetVariable (Fortran), mexGetVariablePtr (Fortran), mexPutVariable (Fortran), global

## Purpose

Determine whether MEX-file is locked

## C Syntax

```
#include "mex.h"
bool mexIsLocked(void);
```

## Returns

Logical `1` (`true`) if the MEX-file is locked; logical `0` (`false`) if the file is unlocked.

## Description

Call `mexIsLocked` to determine if the MEX-file is locked. By default, MEX-files are unlocked, meaning that users can clear the MEX-file at any time.

To unlock a MEX-file, call `mexUnlock`.

## Examples

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

## See Also

`mexLock (C)`, `mexMakeArrayPersistent (C)`, `mexMakeMemoryPersistent (C)`, `mexUnlock (C)`

# mexIsLocked (Fortran)

## Purpose

Determine whether MEX-file is locked

## Fortran Syntax

```
integer*4 function mexIsLocked()
```

## Arguments

None

## Returns

Logical `1` (`true`) if the MEX-file is locked; logical `0` (`false`) if the file is unlocked.

## Description

Call `mexIsLocked` to determine if the MEX-file is locked. By default, MEX-files are unlocked, meaning that users can clear the MEX-file at any time.

To unlock a MEX-file, call `mexUnlock`.

## See Also

`mexLock (Fortran)`, `mexUnlock (Fortran)`, `mexMakeArrayPersistent (Fortran)`, `mexMakeMemoryPersistent (Fortran)`

## Purpose

Prevent MEX-file from being cleared from memory

## C Syntax

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

## Description

By default, MEX-files are unlocked, meaning that a user can clear them at any time. Call `mexLock` to prohibit a MEX-file from being cleared.

To unlock a MEX-file, call `mexUnlock`.

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

## Examples

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

## See Also

`mexIsLocked (C)`, `mexMakeArrayPersistent (C)`, `mexMakeMemoryPersistent (C)`, `mexUnlock (C)`

# mexLock (Fortran)

## Purpose

Prevent MEX-file from being cleared from memory

## Fortran Syntax

```
subroutine mexLock()
```

## Arguments

None

## Description

By default, MEX-files are unlocked, meaning that a user can clear them at any time. Call `mexLock` to prohibit a MEX-file from being cleared.

To unlock a MEX-file, call `mexUnlock`.

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

## See Also

`mexIsLocked (Fortran)`, `mexMakeArrayPersistent (Fortran)`, `mexMakeMemoryPersistent (Fortran)`, `mexUnlock (Fortran)`

## Purpose

Make mxArray persist after MEX-file completes

## C Syntax

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

## Arguments

array_ptr
: Pointer to an mxArray created by an mxCreate* routine.

## Description

By default, mxArrays allocated by mxCreate* routines 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 will leak 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 (C), mexLock (C), mexMakeMemoryPersistent (C), and the mxCreate functions

# mexMakeArrayPersistent (Fortran)

## Purpose

Make mxArray persist after MEX-file completes

## Fortran Syntax

```
subroutine mexMakeArrayPersistent(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray created by an mxCreate* routine.

## Description

By default, mxArrays allocated by mxCreate* routines are not persistent. The MATLAB memory management facility automatically frees nonpersistent mxArrays when the MEX-file finishes. If you want the mxArray to persist through multiple invocations of the MEX-file, 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 will leak memory. See mexAtExit on how to register a function that gets called when the MEX-file is cleared. See mexLock on how to lock your MEX-file so that it is never cleared.

---

## See Also

mexAtExit (Fortran), mexLock (Fortran), mexMakeMemoryPersistent (Fortran), and the mxCreate functions.

## Purpose

Make allocated memory MATLAB persist after MEX-function completes

## C Syntax

```
#include "mex.h"
void mexMakeMemoryPersistent(void *ptr);
```

## Arguments

`ptr`

Pointer to the beginning of memory allocated by one of the MATLAB memory allocation routines.

## Description

By default, memory allocated by MATLAB is nonpersistent, so it is freed automatically when the MEX-file finishes. If you want the memory to persist, you must call `mexMakeMemoryPersistent`.

---

**Note** If you create persistent memory, you are responsible for freeing it when the MEX-function is cleared. If you do not free the memory, MATLAB will leak memory. To free memory, use `mxFree`. See `mexAtExit` to see how to register a function that gets called when the MEX-function is cleared. See `mexLock` to see how to lock your MEX-function so that it is never cleared.

---

## See Also

`mexAtExit (C)`, `mexLock (C)`, `mexMakeArrayPersistent (C)`, `mxCalloc (C)`, `mxFree (C)`, `mxMalloc (C)`, `mxRealloc (C)`

# mexMakeMemoryPersistent (Fortran)

## Purpose

Make allocated memory persist after MEX-file completes

## Fortran Syntax

```
subroutine mexMakeMemoryPersistent(ptr)
MWPOINTER ptr
```

## Arguments

`ptr`

Pointer to the beginning of memory allocated by one of the MATLAB memory allocation routines.

## Description

By default, memory allocated by MATLAB is nonpersistent, so it is freed automatically when the MEX-file finishes. If you want the memory to persist, you must call `mexMakeMemoryPersistent`.

---

**Note** If you create persistent memory, you are responsible for freeing it when the MEX-file is cleared. If you do not free the memory, MATLAB will leak memory. To free memory, use `mxFree`. See `mexAtExit` on how to register a function that gets called when the MEX-file is cleared. See `mexLock` on how to lock your MEX-file so that it is never cleared.

---

## See Also

`mexAtExit (Fortran)`, `mexLock (Fortran)`, `mexMakeArrayPersistent (Fortran)`, `mxCalloc (Fortran)`, `mxFree (Fortran)`, `mxMalloc (Fortran)`, `mxRealloc (Fortran)`

## Purpose

ANSI C `printf`-style output routine

## C Syntax

```
#include "mex.h"
int mexPrintf(const char *format, ...);
```

## Arguments

`format`, ...
ANSI C `printf`-style format string and optional arguments.

## Returns

The number of characters printed. This includes characters specified with backslash codes, such as `\n` and `\b`.

## Description

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.

In a MEX-file, you must call `mexPrintf` instead of `printf`.

## Examples

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

## See Also

`mexErrMsgTxt (C)`, `mexWarnMsgTxt (C)`

# mexPrintf (Fortran)

## Purpose

Print `character` array

## Fortran Syntax

```
integer*4 function mexPrintf(message)
character*(*) message
```

## Arguments

`message`
Character array containing message to be displayed.

---

**Note** Optional arguments to `mexPrintf`, such as format strings, are not supported in Fortran.

---

---

**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.

---

## Returns

The number of characters printed. This includes characters specified with backslash codes, such as `\n` and `\b`.

## Description

`mexPrintf` prints a `character` array 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.

## See Also

`mexErrMsgTxt (Fortran)`

## Purpose

Copy mxArray from MEX-function into specified workspace

## C Syntax

```
#include "mex.h"
int mexPutVariable(const char *workspace, const char *var_name,
const mxArray *array_ptr);
```

## Arguments

workspace
: Specifies the scope of the array that you are copying. The possible values are

| | |
|---|---|
| base | Copy mxArray to the base workspace |
| caller | Copy mxArray to the caller's workspace |
| global | Copy mxArray to the list of global variables |

var_name
: Name given to the mxArray in the workspace.

array_ptr
: Pointer to the mxArray.

## Returns

0 on success; 1 on failure. A possible cause of failure is that array_ptr is NULL.

## Description

Call mexPutVariable to copy the mxArray, at pointer array_ptr, from your MEX-function into the specified workspace. MATLAB gives the name, var_name, to the copied mxArray in the receiving workspace.

mexPutVariable makes the array accessible to other entities, such as MATLAB, M-files or other MEX-functions.

If a variable of the same name already exists in the specified workspace, mexPutVariable overwrites the previous contents of the variable with 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", array_ptr)
```

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

## Examples

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

## See Also

`mexGetVariable (C)`

## Purpose

Copy mxArray into specified workspace

## Fortran Syntax

```
integer*4 function mexPutVariable(workspace, varname, pm)
character*(*) workspace, varname
MWPOINTER pm
```

## Arguments

workspace
: Specifies the scope of the array that you are copying. The possible values are

| | |
|---|---|
| base | Copy the mxArray to the base workspace |
| caller | Copy the mxArray to the caller's workspace |
| global | Copy the mxArray to the list of global variables |

varname
: Name given to the mxArray in the workspace.

pm
: Pointer to an mxArray.

## Returns

0 on success; 1 on failure. A possible cause of failure is that the pm argument is zero.

## Description

Call mexPutVariable to copy the mxArray, at pointer pm, from your MEX-file into the specified workspace. MATLAB gives the name, varname, to the copied mxArray in the receiving workspace.

mexPutVariable makes the array accessible to other entities, such as MATLAB, M-files or other MEX-files.

If a variable of the same name already exists in the specified workspace, mexPutVariable overwrites the previous contents of the variable with 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 MATLAB workspace.

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

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

## See Also

`mexGetVariable (Fortran)`

## 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 (C)`

# mexSetTrapFlag (C)

## Purpose

Control response of `mexCallMATLAB` to errors

## C Syntax

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

## Arguments

`trap_flag`
Control flag. Currently, the only legal 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 `trap_flag` set to 0 is equivalent to not calling `mexSetTrapFlag` at all.

If you call `mexSetTrapFlag` and set the `trap_flag` 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 `trap_flag` 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 `trap_flag` in the first MEX-file is saved.
2. The second MEX-file is called with the `trap_flag` initialized to 0 within that file.

3 When the second MEX-file exits, the saved value of the `trap_flag` in the first MEX-file is restored within that file.

## Examples

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

## See Also

`mexAtExit (C)`, `mexErrMsgTxt (C)`

# mexSetTrapFlag (Fortran)

## Purpose

Control response of `mexCallMATLAB` to errors

## Fortran Syntax

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

## Arguments

`trapflag`

Control flag. Currently, the only legal 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 `trap_flag` 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 `trap_flag` in the first MEX-file is saved.
2. The second MEX-file is called with the `trap_flag` initialized to 0 within that file.

3 When the second MEX-file exits, the saved value of the `trap_flag` in the first MEX-file is restored within that file.

## See Also

`mexAtExit (Fortran)`, `mexErrMsgTxt (Fortran)`

# mexUnlock (C)

## Purpose

Allow MEX-file to be cleared from memory

## C Syntax

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

## 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.

## Examples

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

## See Also

`mexIsLocked (C)`, `mexLock (C)`, `mexMakeArrayPersistent (C)`, `mexMakeMemoryPersistent (C)`

## Purpose

Allow MEX-file to be cleared from memory

## Fortran Syntax

```
subroutine mexUnlock()
```

## Arguments

none

## 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.

## See Also

`mexIsLocked (Fortran)`, `mexLock (Fortran)`, `mexMakeArrayPersistent (Fortran)`, `mexMakeMemoryPersistent (Fortran)`

# mexWarnMsgIdAndTxt (C)

## Purpose

Issue warning message with identifier

## C Syntax

```
#include "mex.h"
void mexWarnMsgIdAndTxt(const char *identifier,
const char *warning_msg, ...);
```

## Arguments

`identifier`
: String containing a MATLAB message identifier. See Message Identifiers in the MATLAB documentation for information on this topic.

`warning_msg`
: String containing the warning message to be displayed. The string may include formatting conversion characters, such as those used with the ANSI C `sprintf` function.

...
: Any additional arguments needed to translate formatting conversion characters used in `warning_msg`. Each conversion character in `warning_msg` is converted to one of these values.

## Description

Call `mexWarnMsgIdAndTxt` to write a warning message and its corresponding identifier to the MATLAB window.

Unlike `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt` does not cause the MEX-file to terminate.

## See Also

`mexWarnMsgTxt (C)`, `mexErrMsgIdAndTxt (C)`, `mexErrMsgTxt (C)`

## Purpose

Issue warning message with identifier

## Fortran Syntax

```
subroutine mexWarnMsgIdAndTxt(warningid, warningmsg)
character*(*) warningid, warningmsg
```

## Arguments

`warningid`
: Character array containing a MATLAB message identifier. See Message Identifiers in the MATLAB documentation for information on this topic.

`warningmsg`
: String containing the warning message to be displayed.

## Description

`mexWarnMsgIdAndTxt` causes MATLAB to display the contents of `warningmsg`.

Unlike `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt` does not cause the MEX-file to terminate.

## See Also

`mexWarnMsgTxt (Fortran)`, `mexErrMsgIdAndTxt (Fortran)`, `mexErrMsgTxt (Fortran)`

# mexWarnMsgTxt (C)

## Purpose

Issue warning message

## C Syntax

```
#include "mex.h"
void mexWarnMsgTxt(const char *warning_msg);
```

## Arguments

`warning_msg`
String containing the warning message to be displayed.

## Description

`mexWarnMsgTxt` causes MATLAB to display the contents of `warning_msg`.

Unlike `mexErrMsgTxt`, `mexWarnMsgTxt` does not cause the MEX-file to terminate.

## Examples

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

For additional examples, see `explore.c` in the `mex` subdirectory of the `examples` directory; see `fulltosparse.c` and `revord.c` in the `refbook` subdirectory of the `examples` directory; see `mxisfinite.c` and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mexWarnMsgIdAndTxt (C)`, `mexErrMsgTxt (C)`, `mexErrMsgIdAndTxt (C)`

## Purpose

Issue warning message

## Fortran Syntax

```
subroutine mexWarnMsgTxt(warningmsg)
character*(*) warningmsg
```

## Arguments

`warningmsg`
String containing the warning message to be displayed.

## Description

`mexWarnMsgTxt` causes MATLAB to display the contents of `warningmsg`.

Unlike `mexErrMsgTxt`, `mexWarnMsgTxt` does not cause the MEX-file to terminate.

## See Also

`mexWarnMsgIdAndTxt (Fortran)`, `mexErrMsgTxt (Fortran)`, `mexErrMsgIdAndTxt (Fortran)`

# MWPOINTER (Fortran)

## 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 `mxArray` or to other data that is not of a native Fortran type, such as memory allocated by `mxMalloc (Fortran)`. 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 (Fortran)` 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.

## Purpose

Add field to structure array

## C Syntax

```
#include "matrix.h"
extern int mxAddField(mxArray array_ptr, const char *field_name);
```

## Arguments

`array_ptr`
Pointer to a structure `mxArray`.

`field_name`
The name of the field you want to add.

## Returns

Field number on success or -1 if inputs are invalid or an out of memory condition occurs.

## Description

Call `mxAddField` to add a field to a structure array. You must then create the values with the `mxCreate*` functions and use `mxSetFieldByNumber` to set the individual values for the field.

## See Also

`mxRemoveField (C)`, `mxSetFieldByNumber (C)`

# mxAddField (Fortran)

## Purpose

Add field to structure array

## Fortran Syntax

```
integer*4 function mxAddField(pm, fieldname)
MWPOINTER pm
character*(*) fieldname
```

## Arguments

`pm`
Pointer to a structure `mxArray`.

`fieldname`
The name of the field you want to add.

## Returns

Field number on success, or `0` if inputs are invalid or an out-of-memory condition occurs.

## Description

Call `mxAddField` to add a field to a structure array. You must then create the values with the `mxCreate*` functions and use `mxSetFieldByNumber` to set the individual values for the field.

## See Also

`mxRemoveField (Fortran)`, `mxSetFieldByNumber (Fortran)`

## Purpose

Convert array to string

## C Syntax

```
#include "matrix.h"
char *mxArrayToString(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to a string `mxArray`; that is, a pointer to an `mxArray` having the `mxCHAR_CLASS` class.

## Returns

A C-style string. Returns NULL on out of memory.

## Description

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.

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

- It does not require the length of the string as an input.
- It supports multibyte character sets.

`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.

## Examples

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

For additional examples, see `mxcreatecharmatrixfromstr.c` and `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxCreateCharArray (C)`, `mxCreateCharMatrixFromStrings (C)`, `mxCreateString (C)`, `mxGetString (C)`

# mxAssert (C)

## Purpose

Check assertion value for debugging purposes

## C Syntax

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

## Arguments

`expr`
: Value of assertion.

`error_message`
: Description of why assertion failed.

## Description

Similar to the ANSI C `assert()` macro, `mxAssert` checks the value of an assertion, and continues execution only if the assertion holds. If `expr` evaluates to logical `1` (`true`), `mxAssert` does nothing. If `expr` evaluates to logical `0` (`false`), `mxAssert` 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 `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 you should use this for debugging purposes only. Build the mex file using the syntax, `mex -g filename`, in order to use `mxAssert`.

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.

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.

## 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

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 you should use this for debugging purposes only. Build the mex file using the syntax, `mex -g filename`, in order to use `mxAssert`.

# mxCalcSingleSubscript (C)

## Purpose

Offset from first element to desired element

## C Syntax

```
#include <matrix.h>
int mxCalcSingleSubscript(const mxArray *array_ptr, int nsubs,
    int *subs);
```

## Arguments

`array_ptr`
: 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 `array_ptr` points to.

subs
: An array of integers. Each value in the array should specify that dimension's subscript. The value in `subs[0]` specifies the row subscript, and the value in `subs[1]` specifies the column subscript. Note that `mxCalcSingleSubscript` views 0 as the first element of an `mxArray`, but MATLAB sees 1 as the first element of an `mxArray`. For example, in MATLAB, `(1,1)` denotes the starting element of a two-dimensional `mxArray`; however, to express the starting element of a two-dimensional `mxArray` in `subs`, you must set `subs[0]` to `0` and `subs[1]` to `0`.

## 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.

If `subs` describes the starting element of an `mxArray`, `mxCalcSingleSubscript` returns 0. If `subs` describes the final element of an `mxArray`, then `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 `mxArray` and a given element of that `mxArray`. For example, given a subscript like `(5,7)`,

`mxCalcSingleSubscript` returns the distance from the `(0,0)` element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have.

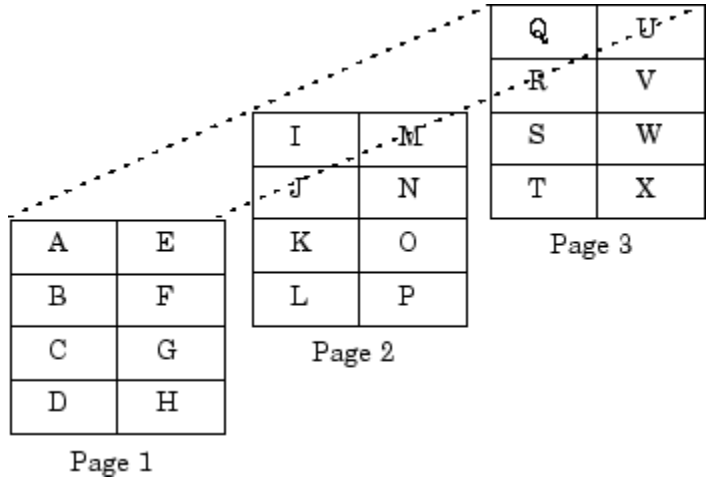
MATLAB uses a column-major numbering scheme to represent data elements internally. That means that MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on through the last column. For example, suppose you create a 4-by-2 variable.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

| | |
|---|---|
| A | E |
| B | F |
| C | G |
| D | H |

Although in fact, 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 `mxArray` is N-dimensional, then 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




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

| 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. 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`.

## Examples

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

## Purpose

Offset from first element to desired element

## Fortran Syntax

```
integer*4 function mxCalcSingleSubscript(pm, nsubs, subs)
MWPOINTER pm
integer*4 nsubs, 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. The value in `subs(1)` specifies the row subscript, and the value in `subs(2)` specifies the column subscript. Use 1-based indexing to specify the desired array element. 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.

If `subs` describes the starting element of an `mxArray`, `mxCalcSingleSubscript` returns 0. If `subs` describes the final element of an `mxArray`, then `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 `mxArray` and a given element of that `mxArray`. For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the `(1,1)` element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have.

Use `mxCalcSingleSubscript` with functions that interact with multidimensional cells and structures. `mxGetCell (Fortran)` and `mxSetCell (Fortran)` are two such functions.

## See Also

`mxGetCell (Fortran)`, `mxSetCell (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);
```

## 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 (nonMEX-file) application, `mxCalloc` returns `NULL`. 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 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.

## Examples

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

`mxFree (C)`, `mxDestroyArray (C)`, `mexMakeArrayPersistent (C)`, `mexMakeMemoryPersistent (C)`, `mxMalloc (C)`, `mxRealloc (C)`

## Purpose

Allocate dynamic memory for array using MATLAB memory manager

## Fortran Syntax

```
MWPOINTER function mxCalloc(n, size)
integer*4 n, size
```

## Arguments

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

`size`
: Number of bytes per element.

## Returns

A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCalloc` returns 0. 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

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

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

`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.

In stand-alone MATLAB applications, the MATLAB memory manager is not used.

## See Also

`mxFree (Fortran)`, `mxMalloc (Fortran)`, `mxRealloc (Fortran)`

## Purpose

Data type for string mxArray

## C Syntax

```
typedef Uint16 mxChar;
```

## Description

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.

## Examples

See mxmalloc.c in the mx subdirectory of the examples directory.

For additional examples, see explore.c in the mex subdirectory of the examples directory and mxcreatecharmatrixfromstr.c in the mx subdirectory of the examples directory.

## See Also

mxCreateCharArray (C)

## 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`; that is, an `mxArray` that stores Boolean elements logical `1` (`true`) and logical `0` (`false`).

`mxCHAR_CLASS`
: Identifies a string `mxArray`; that is 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 (C)`

## 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 table below.

## 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 (Fortran)` and `mxCreateNumericMatrix (Fortran)`.

Valid choices for `classname` are shown below. 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 (Fortran)`, `mxCreateNumericArray (Fortran)`, `mxCreateNumericMatrix (Fortran)`

# mxComplexity (C)

## Purpose

Flag specifying whether mxArray has imaginary components

## C Syntax

```
typedef enum mxComplexity {mxREAL=0, mxCOMPLEX};
```

## Constants

mxREAL
: Identifies an mxArray with no imaginary components.

mxCOMPLEX
: Identifies an mxArray with imaginary components.

## Description

Various mx calls require an mxComplexity argument. You can set an mxComplex argument to either mxREAL or mxCOMPLEX.

## Examples

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

## See Also

mxCreateNumericArray (C), mxCreateDoubleMatrix (C), mxCreateSparse (C)

## Purpose

Copy character values from Fortran array to pointer array

## Fortran Syntax

```
subroutine mxCopyCharacterToPtr(y, px, n)
character*(*) y
MWPOINTER px
integer*4 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 (Fortran), mxCreateCharArray (Fortran), mxCreateString (Fortran), mxCreateCharMatrixFromStrings (Fortran)

# 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
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (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
integer*4 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 (Fortran)`, `mxCreateNumericArray (Fortran)`, `mxCreateNumericMatrix (Fortran)`, `mxGetData (Fortran)`, `mxGetImagData (Fortran)`

# mxCopyInteger1ToPtr (Fortran)

## Purpose

Copy INTEGER*1 values from Fortran array to pointer array

## Fortran Syntax

```
subroutine mxCopyInteger1ToPtr(y, px, n)
integer*1 y(n)
MWPOINTER px
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (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
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran)

# mxCopyInteger4ToPtr (Fortran)

## Purpose

Copy INTEGER*4 values from Fortran array to pointer array

## Fortran Syntax

```
subroutine mxCopyInteger4ToPtr(y, px, n)
integer*4 y(n)
MWPOINTER px
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran)

## Purpose

Copy character values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToCharacter(px, y, n)
MWPOINTER px
character*(*) y
integer*4 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 (Fortran)`, `mxCreateCharArray (Fortran)`, `mxCreateString (Fortran)`, `mxCreateCharMatrixFromStrings (Fortran)`

# mxCopyPtrToComplex16 (Fortran)

## Purpose

Copy COMPLEX*16 values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToComplex16(pr, pi, y, n)
MWPOINTER pr, pi
complex*16 y(n)
integer*4 n
```

## Arguments

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

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

`y`
: COMPLEX*16 Fortran array.

`n`
: Number of elements to copy.

## Description

mxCopyPtrToComplex16 copies n COMPLEX*16 values from the MATLAB arrays pointed to by pr and pi into the Fortran COMPLEX*16 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.

## See Also

mxCopyComplex16ToPtr (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (Fortran)

## Purpose

Copy COMPLEX*8 values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToComplex8(pr, pi, y, n)
MWPOINTER pr, pi
complex*8 y(n)
integer*4 n
```

## Arguments

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

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

`y`
: COMPLEX*8 Fortran array.

`n`
: Number of elements to copy.

## Description

mxCopyPtrToComplex8 copies n COMPLEX*8 values from the MATLAB arrays pointed to by pr and pi into the Fortran COMPLEX*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.

## See Also

mxCopyComplex8ToPtr (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (Fortran)

# mxCopyPtrToInteger1 (Fortran)

## Purpose

Copy INTEGER*1 values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToInteger1(px, y, n)
MWPOINTER px
integer*1 y(n)
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (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)
integer*4 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 (Fortran)`, `mxCreateNumericArray (Fortran)`, `mxCreateNumericMatrix (Fortran)`

# mxCopyPtrToInteger4 (Fortran)

## Purpose

Copy INTEGER*4 values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToInteger4(px, y, n)
MWPOINTER px
integer*4 y(n)
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran)

## Purpose

Copy pointer values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToPtrArray(px, y, n)
MWPOINTER px
integer*4 y(n)
integer*4 n
```

## Arguments

`px`
: Pointer to pointer array.

`y`
: `INTEGER*4` Fortran array.

`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 (Fortran)`, `mxCopyPtrToCharacter (Fortran)`

# mxCopyPtrToReal4 (Fortran)

## Purpose

Copy REAL*4 values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToReal4(px, y, n)
MWPOINTER px
real*4 y(n)
integer*4 n
```

## Arguments

px
: Pointer to the real or imaginary data of a single-precision MATLAB array.

y
: REAL*4 Fortran array.

n
: Number of elements to copy.

## Description

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.

## See Also

mxCopyReal4ToPtr (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (Fortran)

## Purpose

Copy REAL*8 values from pointer array to Fortran array

## Fortran Syntax

```
subroutine mxCopyPtrToReal8(px, y, n)
MWPOINTER px
real*8 y(n)
integer*4 n
```

## Arguments

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

y
: REAL*8 Fortran array.

n
: Number of elements to copy.

## Description

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.

## Examples

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.

## See Also

mxCopyReal8ToPtr (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (Fortran)

# mxCopyReal4ToPtr (Fortran)

## Purpose

Copy REAL*4 values from Fortran array to pointer array

## Fortran Syntax

```
subroutine mxCopyReal4ToPtr(y, px, n)
real*4 y(n)
MWPOINTER px
integer*4 n
```

## Arguments

y
: REAL*4 Fortran array.

px
: Pointer to the real or imaginary data of a single-precision MATLAB array.

n
: Number of elements to copy.

## Description

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.

## See Also

mxCopyPtrToReal4 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (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
integer*4 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 (Fortran), mxCreateNumericArray (Fortran), mxCreateNumericMatrix (Fortran), mxGetData (Fortran), mxGetImagData (Fortran)

# mxCreateCellArray (C)

## Purpose

Create unpopulated N-D cell mxArray

## C Syntax

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

## Arguments

ndim
: The desired number of dimensions in the created cell. For example, to create a three-dimensional cell mxArray, set ndim to 3.

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

## Returns

A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellArray returns NULL. 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.

## Description

Use mxCreateCellArray to create a cell mxArray whose size is defined by ndim and dims. For example, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set

```
ndim = 3;
dims[0] = 4; dims[1] = 8; dims[2] = 7;
```

The created cell mxArray is unpopulated; that is, 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.

## Examples

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

## See Also

`mxCreateCellMatrix (C)`, `mxGetCell (C)`, `mxSetCell (C)`, `mxIsCell (C)`

# mxCreateCellArray (Fortran)

## Purpose

Create unpopulated N-D cell mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateCellArray(ndim, dims)
integer*4 ndim, dims
```

## Arguments

ndim

The desired number of dimensions in the created cell. For example, to create a three-dimensional cell mxArray, set ndim to 3.

dims

The dimensions array. Each element in the dimensions array contains the size of the mxArray in that dimension. For example, 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.

## Returns

A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellArray returns 0. 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.

## Description

Use mxCreateCellArray to create a cell mxArray whose size is defined by ndim and dims. For example, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set

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

The created cell mxArray is unpopulated; that is, mxCreateCellArray initializes each cell to 0. 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.

## See Also

mxCreateCellMatrix (Fortran), mxGetCell (Fortran), mxSetCell (Fortran), mxIsCell (Fortran)

# mxCreateCellMatrix (C)

## Purpose

Create unpopulated 2-D cell mxArray

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateCellMatrix(int m, int n);
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

## Returns

A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellMatrix returns NULL. 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.

## Description

Use mxCreateCellMatrix to create an m-by-n two-dimensional cell mxArray. The created cell mxArray is unpopulated; that is, mxCreateCellMatrix initializes each cell to NULL. To put data into cells, call mxSetCell.

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.

## Examples

See mxcreatecellmatrix.c in the mx subdirectory of the examples directory.

## See Also

mxCreateCellArray (C)

## Purpose

Create unpopulated 2-D cell mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateCellMatrix(m, n)
integer*4 m, n
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

## Returns

A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellMatrix returns 0. 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.

## Description

Use mxCreateCellMatrix to create an m-by-n two-dimensional cell mxArray. The created cell mxArray is unpopulated; that is, mxCreateCellMatrix initializes each cell to 0. To put data into the cells, call mxSetCell.

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.

## See Also

mxCreateCellArray (Fortran)

# mxCreateCharArray (C)

## Purpose

Create unpopulated N-D string mxArray

## C Syntax

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

## Arguments

`ndim`

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.

`dims`

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. The dims array must have at least ndim elements.

## Returns

A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCharArray returns NULL. 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.

## Description

Call mxCreateCharArray to create an unpopulated N-dimensional string mxArray.

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.

## Examples

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

## See Also

mxCreateCharMatrixFromStrings (C), mxCreateString (C)

## Purpose

Create unpopulated N-D character mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateCharArray(ndim, dims)
integer*4 ndim, dims
```

## Arguments

ndim

The desired number of dimensions in the character mxArray. You must specify a positive number. If you specify 0, 1, or 2, mxCreateCharArray 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, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 character mxArray. The dims array must have at least ndim elements.

## Returns

A pointer to the created character mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCharArray returns 0. 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.

## Description

Use mxCreateCharArray to create an mxArray of characters whose size is defined by ndim and dims. For example, to establish a two-dimensional mxArray of characters having dimensions 12-by-3, set

```
ndim = 2;
dims(1) = 12; dims(2) = 3;
```

The created mxArray is unpopulated; that is, mxCreateCharArray initializes each character to INTEGER*2 0.

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.

**See Also** mxCreateString (Fortran)

## Purpose

Create populated 2-D string mxArray

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateCharMatrixFromStrings(int m, const char **str);
```

## Arguments

`m`

The desired number of rows in the created string mxArray. The value you specify for m should equal the number of strings in str.

`str`

A pointer to a list of strings. The str array must contain at least m strings.

## Returns

A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCharMatrixFromStrings returns NULL. 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 str contains fewer than m strings.

## Description

Use mxCreateCharMatrixFromStrings to create a two-dimensional string mxArray, where each row is initialized to a string from str. The created mxArray has dimensions m-by-max, where max is the length of the longest string in str.

Note that string mxArrays represent their data elements as mxChar rather than as char.

## Examples

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

## See Also

mxCreateCharArray (C), mxCreateString (C), mxGetString (C)

# mxCreateCharMatrixFromStrings (Fortran)

## Purpose

Create populated 2-D char mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateCharMatrixFromStrings(m, str)
integer*4 m
character*(*) str(m)
```

## Arguments

m

The desired number of rows in the created string mxArray. The value you specify for m should equal the size of the str array.

str

A Fortran character*n array of size m, where each element of the array is n bytes.

## Returns

A pointer to the created char mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCharMatrixFromStrings returns 0. 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 mxCreateCharMatrixFromStrings to be unsuccessful. Another possible reason for failure is that str contains fewer than m strings.

## Description

Use mxCreateCharMatrixFromStrings to create a two-dimensional string mxArray, where each row is initialized to str. The created mxArray has dimensions m-by-n, where n is the length of the number of characters in str(i).

## See Also

mxCreateCharArray (Fortran), mxCreateString (Fortran)

## Purpose

Create unpopulated 2-D, double-precision, floating-point mxArray

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateDoubleMatrix(int m, int n,
mxComplexity ComplexFlag);
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

ComplexFlag
: Specify either mxREAL or mxCOMPLEX. If the data you plan to put into the mxArray has no imaginary components, specify mxREAL. If the data has some imaginary components, specify mxCOMPLEX.

## Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateDoubleMatrix returns NULL. 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.

## Description

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, mxCreateDoubleMatrix also initializes each element in the pi array to 0.

If you set ComplexFlag to mxREAL, mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements. If you set ComplexFlag to mxCOMPLEX, 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.

## 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 (C)`, `mxComplexity (C)`

## Purpose

Create unpopulated 2-D, double-precision, floating-point mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateDoubleMatrix(m, n, ComplexFlag)
integer*4 m, n, ComplexFlag
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

ComplexFlag
: If the data you plan to put into the mxArray has no imaginary component, specify 0. If the data has some imaginary components, specify 1.

## Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateDoubleMatrix returns 0. 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.

## Description

Use mxCreateDoubleMatrix to create an m-by-n mxArray.

If you set ComplexFlag to 0, mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements and initializes each element to 0.0.

If you set ComplexFlag to 1, mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements and m-by-n imaginary elements. It initializes each real and imaginary element to 0.0.

Call mxDestroyArray when you finish using the mxArray. mxDestroyArray deallocates the mxArray and its associated real and complex elements.

## See Also

mxCreateNumericArray (Fortran)

# mxCreateDoubleScalar (C)

## Purpose

Create scalar, double-precision array initialized to specified value

## C Syntax

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

## Description

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

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

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

## See Also

`mxGetPr (C)`, `mxCreateDoubleMatrix (C)`

## Purpose

Create scalar, double-precision array initialized to specified 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 `0`.

## Description

Call `mxCreateDoubleScalar` to create a scalar double `mxArray`. `mxCreateDoubleScalar` is a convenience function that can be used in place of the following 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 (Fortran)`, `mxCreateDoubleMatrix (Fortran)`

# mxCreateLogicalArray (C)

## Purpose

Create N-D logical mxArray initialized to false

## C Syntax

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

## Arguments

ndim

Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateLogicalArray automatically sets the number of dimensions to 2.

dims

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.

## Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-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.

## Description

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.

mxCreateLogicalArray 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.

## See Also

mxCreateLogicalMatrix (C), mxCreateSparseLogicalMatrix (C), mxCreateLogicalScalar (C)

# mxCreateLogicalMatrix (C)

## Purpose

Create 2-D, logical mxArray initialized to false

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalMatrix(int m, int n);
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

## Returns

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.

## Description

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.

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

## See Also

mxCreateLogicalArray (C), mxCreateSparseLogicalMatrix (C), 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 (nonMEX-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

mxIsLogicalScalar (C), mxIsLogicalScalarTrue (C), mxCreateLogicalMatrix (C), mxCreateLogicalArray (C), mxGetLogicals (C)

# mxCreateNumericArray (C)

## Purpose

Create unpopulated N-D numeric mxArray

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateNumericArray(int ndim, const int *dims,
         mxClassID class, mxComplexity ComplexFlag);
```

## Arguments

ndim
: Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateNumericArray automatically sets the number of dimensions to 2.

dims
: 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. In most cases, there should be ndim elements in the dims array.

class
: The way in which the numerical data is to be represented in memory. For example, specifying mxINT16_CLASS causes each piece of numerical data in the mxArray to be represented as a 16-bit signed integer. You can specify any class except for mxNUMERIC_CLASS, mxSTRUCT_CLASS, or mxCELL_CLASS.

ComplexFlag
: Specify either mxREAL or mxCOMPLEX. If the data you plan to put into the mxArray has no imaginary components, specify mxREAL. If the data will have some imaginary components, specify mxCOMPLEX.

## Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateNumericArray returns NULL. 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 class. After creating the mxArray, mxCreateNumericArray initializes all its real data elements to 0. If ComplexFlag equals mxCOMPLEX, 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.

## 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.

## See Also

mxClassID (C), mxCreateDoubleMatrix (C), mxCreateSparse (C), mxCreateString (C), mxComplexity (C)

# mxCreateNumericArray (Fortran)

## Purpose

Create unpopulated N-D numeric mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateNumericArray(ndim, dims, classid,
ComplexFlag)
integer*4 ndim, dims, classid, ComplexFlag
```

## Arguments

ndim

Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateNumericArray automatically sets the number of dimensions to 2.

dims

The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, 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.

classid

A numerical identifier that represents a particular MATLAB class. Use the function mxClassIDFromClassName (Fortran) to derive the classid value from a class name character array.

The classid tells MATLAB how you want the numerical array data to be represented in memory. For example, specifying the int32 class causes each piece of numerical data in the mxArray to be represented as a 32-bit signed integer.

mxCreateNumericArray accepts any of the MATLAB signed numeric classes, shown to the left in the table below.

ComplexFlag

If the data you plan to put into the mxArray has no imaginary components, specify 0. If the data will have some imaginary components, specify 1.

## Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateNumericArray

returns 0. 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 is set to 1, mxCreateNumericArray also initializes all its imaginary data elements to 0.

The following table shows the Fortran data types that are equivalent to MATLAB classes. Use these as shown in the example below.

| MATLAB Class Name | Fortran Type |
|---|---|
| int8 | INTEGER*1 |
| int16 | INTEGER*2 |
| int32 | INTEGER*4 |
| single | REAL*4 |
| double | REAL*8 |
| single, with imaginary components | COMPLEX*8 |
| double, with imaginary components | COMPLEX*16 |

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.

## 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)
```

## See Also

mxCreateDoubleMatrix (Fortran), mxCreateNumericMatrix (Fortran), mxCreateSparse (Fortran), mxCreateString (Fortran)

## Purpose

Create numeric matrix and initialize data elements to 0

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateNumericMatrix(int m, int n, mxClassID class,
mxComplexity ComplexFlag);
```

## Arguments

`m`
: The desired number of rows.

`n`
: The desired number of columns.

`class`
: The way in which the numerical data is to be represented in memory. For example, specifying `mxINT16_CLASS` causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer. You can specify any numeric class including `mxDOUBLE_CLASS`, `mxSINGLE_CLASS`, `mxINT8_CLASS`, `mxUINT8_CLASS`, `mxINT16_CLASS`, `mxUINT16_CLASS`, `mxINT32_CLASS`, `mxUINT32_CLASS`, `mxINT64_CLASS`, and `mxUINT64_CLASS`.

`ComplexFlag`
: Specify either `mxREAL` or `mxCOMPLEX`. If the data you plan to put into the `mxArray` has no imaginary components, specify `mxREAL`. If the data has some imaginary components, specify `mxCOMPLEX`.

## Returns

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

## Description

Call `mxCreateNumericMatrix` to create a 2-D `mxArray` in which all data elements have the numeric data type specified by `class`. After creating the `mxArray`, `mxCreateNumericMatrix` initializes

all its real data elements to 0. If `ComplexFlag` equals `mxCOMPLEX`, `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.

## See Also

`mxCreateNumericArray (C)`

## Purpose

Create numeric matrix and initialize data elements to 0

## Fortran Syntax

```
MWPOINTER function mxCreateNumericMatrix(m, n, classid,
ComplexFlag)
integer*4 m, n, classid, ComplexFlag
```

## Arguments

`m`
: The desired number of rows.

`n`
: The desired number of columns.

`classid`
: A numerical identifier that represents a particular MATLAB class. Use the function `mxClassIDFromClassName (Fortran)` to derive the `classid` value from a class name `character` array.

The `classid` tells MATLAB how you want the numerical array data to be represented in memory. For example, specifying the `int32` class causes each piece of numerical data in the `mxArray` to be represented as a 32-bit signed integer.

`mxCreateNumericMatrix` accepts any of the MATLAB signed numeric classes, shown to the left in the table below.

`ComplexFlag`
: If the data you plan to put into the `mxArray` has no imaginary components, specify `0`. If the data has some imaginary components, specify `1`.

## Returns

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

# mxCreateNumericMatrix (Fortran)

## Description

Call `mxCreateNumericMatrix` to create an two-dimensional `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` is set to `1`, `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 Fortran data types that are equivalent to MATLAB classes. Use these as shown in the example below.

| MATLAB Class Name | Fortran Type |
|---|---|
| `int8` | `BYTE` |
| `int16` | `INTEGER*2` |
| `int32` | `INTEGER*4` |
| `single` | `REAL*4` |
| `double` | `REAL*8` |
| `single`, with imaginary components | `COMPLEX*8` |
| `double`, with imaginary components | `COMPLEX*16` |

## 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

`mxCreateDoubleMatrix (Fortran)`, `mxCreateNumericArray (Fortran)`

## Purpose

Create 2-D unpopulated sparse mxArray

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateSparse(int m, int n, int nzmax,
         mxComplexity ComplexFlag);
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

nzmax
: The number of elements that mxCreateSparse should allocate to hold the pr, ir, and, if ComplexFlag is mxCOMPLEX, 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.

ComplexFlag
: Set this value to mxREAL or mxCOMPLEX. If the mxArray you are creating is to contain imaginary data, then set ComplexFlag to mxCOMPLEX. Otherwise, set ComplexFlag to mxREAL.

## Returns

A pointer to the created sparse double mxArray if successful, and NULL otherwise. The most likely reason for failure is insufficient free heap space. If that happens, try reducing nzmax, m, or n.

## Description

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. In order to make the returned sparse mxArray useful, you must initialize the pr, ir, jc, and (if it exists) pi array.

mxCreateSparse allocates space for

- A pr array of length nzmax.
- A pi array of length nzmax (but only if ComplexFlag is mxCOMPLEX).

- 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.

## Examples

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

## See Also

`mxDestroyArray (C)`, `mxSetNzmax (C)`, `mxSetPr (C)`, `mxSetPi (C)`, `mxSetIr (C)`, `mxSetJc (C)`, `mxComplexity (C)`

## Purpose

Create 2-D unpopulated sparse mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateSparse(m, n, nzmax, ComplexFlag)
integer*4 m, n, nzmax, ComplexFlag
```

## Arguments

m
: The desired number of rows.

n
: The desired number of columns.

nzmax
: The number of elements that mxCreateSparse should allocate to hold the pr, ir, and, if ComplexFlag = 1, 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.

ComplexFlag
: Specify REAL = 0 if the data has no imaginary components; specify COMPLEX = 1 if the data has some imaginary components.

## Returns

An unpopulated, sparse double mxArray if successful, and 0 otherwise.

## Description

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. In order to make the returned sparse mxArray useful, you must initialize the pr, ir, jc, and (if it exists) pi array.

mxCreateSparse allocates space for

- A pr array of length nzmax.
- A pi array of length nzmax (but only if ComplexFlag is COMPLEX = 1).
- 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.

## See Also

mxDestroyArray (Fortran), mxSetNzmax (Fortran), mxSetPr (Fortran), mxSetIr (Fortran), mxSetJc (Fortran)

## Purpose

Create unpopulated 2-D, sparse, logical mxArray

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateSparseLogicalMatrix(int m, int n, int 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

mxCreateLogicalMatrix (C), mxCreateLogicalArray (C), mxCreateLogicalScalar (C), mxCreateSparse (C), mxIsLogical (C)

# mxCreateString (C)

## Purpose

Create 1-by-N string mxArray initialized to specified string

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateString(const char *str);
```

## Arguments

str
: The C string that is to serve as the mxArray's initial data.

## Returns

A pointer to the created string mxArray if successful, and NULL otherwise. The most likely cause of failure is insufficient free heap space.

## Description

Use mxCreateString to create a string mxArray initialized to str. Many MATLAB functions (for example, strcmp and upper) require string array inputs.

Free the string mxArray when you are finished using it. To free a string mxArray, call mxDestroyArray.

## Examples

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

For additional examples, see mxcreatestructarray.c and mxisclass.c in the mx subdirectory of the examples directory.

## See Also

mxCreateCharMatrixFromStrings (C), mxCreateCharArray (C)

## Purpose

Create 1-by-N `character` array initialized to specified string

## Fortran Syntax

```
MWPOINTER function mxCreateString(str)
character*(*) str
```

## Arguments

`str`

The string that is to serve as the `mxArray's` initial data.

## Returns

A `character` array initialized to `str` if successful, and 0 otherwise.

## Description

Use `mxCreateString` to create a `character` `mxArray` initialized to `str`. Many MATLAB functions (for example, `strcmp` and `upper`) require `character` `mxArray` inputs.

Free the `character` `mxArray` when you are finished using it. To free a `character` `mxArray`, call `mxDestroyArray`.

## Examples

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.

## See Also

`mxDestroyArray (Fortran)`

# mxCreateStructArray (C)

## Purpose

Create unpopulated N-D structure mxArray

## C Syntax

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

## 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, setting `dims[0]` to 5 and `dims[1]` 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.

`field_names`
: The desired list of field names.

Structure field names must begin with a letter, and are 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` 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 `field_names`. 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. 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.

## Examples

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

## See Also

mxDestroyArray (C), mxSetNzmax (C), namelengthmax

# mxCreateStructArray (Fortran)

## Purpose

Create unpopulated N-D structure mxArray

## Fortran Syntax

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

## Arguments

`ndim`
: Number of dimensions. If you set `ndim` to be less than 2, `mxCreateStructArray` 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, 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.

Structure field names must begin with a letter, and are 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 zero 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`.

Each field holds one mxArray pointer. mxCreateStructArray initializes each field to zero. Call mxSetField or mxSetFieldByNumber to place a non-zero 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.

## See Also

mxDestroyArray (Fortran), mxCreateStructMatrix (Fortran), mxIsStruct (Fortran), mxAddField (Fortran), mxSetField (Fortran), mxGetField (Fortran), mxRemoveField (Fortran), namelengthmax

# mxCreateStructMatrix (C)

## Purpose

Create unpopulated 2-D structure mxArray

## C Syntax

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

## 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.

field_names
: The desired list of field names.

Structure field names must begin with a letter, and are 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 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 only create two-dimensional mxArrays, while mxCreateStructArray can create mxArrays having two or more dimensions.

## Examples

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

## See Also

mxCreateStructArray (C), mxGetFieldByNumber (C), mxGetFieldNameByNumber (C), mxGetFieldNumber (C), mxIsStruct (C), namelengthmax

## Purpose

Create unpopulated 2-D structure mxArray

## Fortran Syntax

```
MWPOINTER function mxCreateStructMatrix(m, n, nfields, fieldnames)
integer*4 m, n, 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.

Structure field names must begin with a letter, and are 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 `0` 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 only create two-dimensional `mxArrays`, while `mxCreateStructArray` can create `mxArrays` having two or more dimensions.

## See Also

`mxCreateStructArray (Fortran)`, `mxIsStruct (Fortran)`, `mxAddField (Fortran)`, `mxSetField (Fortran)`, `mxGetField (Fortran)`, `mxRemoveField (Fortran)`, `namelengthmax`

# mxDestroyArray (C)

## Purpose

Free dynamic memory allocated by mxCreate

## C Syntax

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

## Arguments

array_ptr
Pointer to the mxArray that 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 mxArray's associated data arrays (such as pr, pi, ir, and/or jc). You should not call mxDestroyArray on an mxArray you are returning on the left-hand side.

## Examples

See sincall.c in the refbook subdirectory of the examples directory.

For additional examples, see mexcallmatlab.c and mexgetarray.c in the mex subdirectory of the examples directory; see mxisclass.c in the mx subdirectory of the examples directory.

## See Also

mxCalloc (C), mxFree (C), mexMakeArrayPersistent (C), mexMakeMemoryPersistent (C)

## Purpose

Free dynamic memory allocated by mxCreate

## Fortran Syntax

```
subroutine mxDestroyArray(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to the mxArray that 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 mxArray's associated data arrays (such as pr, pi, ir, and/or jc). You should not call mxDestroyArray on an mxArray you are returning on the left-hand side.

## See Also

mxCalloc (Fortran), mxFree (Fortran), mexMakeArrayPersistent (Fortran), mexMakeMemoryPersistent (Fortran)

# mxDuplicateArray (C)

## Purpose

Make deep copy of array

## C Syntax

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

## Arguments

`in`

Pointer to the `mxArray` that 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 the each cell (if any), and so on.

## Examples

See `mexget.c` in the `mex` subdirectory of the `examples` directory and `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.

## Purpose

Make deep copy of array

## Fortran Syntax

```
MWPOINTER function mxDuplicateArray(in)
MWPOINTER in
```

## Arguments

`in`

Pointer to the `mxArray` that 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 the each cell (if any), and so on.

# mxFree (C)

## Purpose

Free dynamic memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`

## C Syntax

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

## Arguments

`ptr`

Pointer to the beginning of any memory parcel allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

## Description

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

`mxFree` works differently in MEX-files than in stand-alone MATLAB applications.

In MEX-files, `mxFree` automatically

- Calls the ANSI C `free` function, which deallocates the contiguous heap space that begins at address `ptr`.
- Removes this memory parcel from the MATLAB memory management facility's list of memory parcels.

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

When `mxFree` appears in stand-alone MATLAB applications, `mxFree` simply calls the ANSI C `free` function.

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 `mxCalloc` are nonpersistent. However, if an application calls `mexMakeMemoryPersistent`, then the specified memory parcel becomes persistent.

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

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 clean-up handler. Then, the clean-up handler calls `mxFree`.

## Examples

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

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

## See Also

`mxCalloc (C)`, `mxDestroyArray (C)`, `mxMalloc (C)`, `mxRealloc (C)`, `mexMakeArrayPersistent (C)`, `mexMakeMemoryPersistent (C)`

# mxFree (Fortran)

## Purpose

Free dynamic memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`

## Fortran Syntax

```
subroutine mxFree(ptr)
MWPOINTER ptr
```

## Arguments

`ptr`

Pointer to the beginning of any memory parcel allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

## Description

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

`mxFree` works differently in MEX-files than in stand-alone MATLAB applications. With MEX-files, `mxFree` returns to the heap any memory allocated using `mxCalloc`. If you do not free memory with this command, MATLAB frees it automatically on return from the MEX-file. In stand-alone MATLAB applications, you have to explicitly free memory, and MATLAB memory management is not used.

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 `mxCalloc` 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 a good programming practice to deallocate memory just as soon as you are through using it. Doing so generally makes the entire system run more efficiently.

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 clean-up handler. Then, the clean-up handler calls `mxFree`.

## See Also

mxCalloc (Fortran), mxRealloc (Fortran), mxDestroyArray (Fortran)

# mxGetCell (C)

## Purpose

Contents of mxArray cell

## C Syntax

```
#include "matrix.h"
mxArray *mxGetCell(const mxArray *array_ptr, int index);
```

## Arguments

`array_ptr`
Pointer to a cell `mxArray`.

`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.

## Returns

A pointer to the `i`th cell `mxArray` if successful, and `NULL` otherwise. Causes of failure include

- The indexed cell array element has not been populated.
- Specifying an `array_ptr` that does not point to a cell `mxArray`.
- Specifying an `index` greater than the number of elements in the cell.
- Insufficient free heap space to hold the returned cell `mxArray`.

## Description

Call `mxGetCell` to get a pointer to the `mxArray` held in the indexed element of the 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.

---

## Examples

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

## See Also

`mxCreateCellArray (C)`, `mxIsCell (C)`, `mxSetCell (C)`

## Purpose

Contents of cell

## Fortran Syntax

```
MWPOINTER function mxGetCell(pm, index)
MWPOINTER pm
integer*4 index
```

## Arguments

`pm`

Pointer to a cell `mxArray`.

`index`

The number of elements in the cell `mxArray` between the first element and the desired one. See `mxCalcSingleSubscript (Fortran)` for details on calculating an index in a multidimensional cell array.

## Returns

A pointer to the `i`th cell `mxArray` if successful, and `0` otherwise. Causes of failure include

- The indexed cell array element has not been populated.
- Specifying an array pointer, `pm`, that does not point to a cell `mxArray`.
- Specifying an `index` greater than the number of elements in the cell.
- Insufficient free heap space to hold the returned cell `mxArray`.

## Description

Call `mxGetCell` to get a pointer to the `mxArray` held in the indexed element of the 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.

---

## See Also

`mxCreateCellArray (Fortran)`, `mxIsCell (Fortran)`, `mxSetCell (Fortran)`

# mxGetChars (C)

## Purpose

Pointer to character array data

## C Syntax

```
#include "matrix.h"
mxChar *mxGetChars(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

The address of the first character in the `mxArray`. Returns `NULL` if the specified array is not a character array.

## Description

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`.

## See Also

`mxGetString (C)`, `mxGetPr (C)`, `mxGetPi (C)`, `mxGetCell (C)`, `mxGetField (C)`, `mxGetLogicals (C)`, `mxGetScalar (C)`

## Purpose

Class of `mxArray`

## C Syntax

```
#include "matrix.h"
mxClassID mxGetClassID(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
: Pointer to an `mxArray`.

## Returns

The class (category) of the `mxArray` that `array_ptr` points to. Classes are

`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`.

`mxCHAR_CLASS`
: Identifies a string `mxArray`; that is an `mxArray` whose data is represented as `mxCHAR`'s.

`mxLOGICAL_CLASS`
: Identifies a logical `mxArray`; that is, an `mxArray` that stores the logical values `1` and `0`, representing the states `true` and `false` respectively.

`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

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 `array_ptr` points to a logical `mxArray`, then `mxGetClassID` returns `mxLOGICAL_CLASS`.

`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.

## Examples

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

## See Also

`mxGetClassName (C)`

# mxGetClassID (Fortran)

## Purpose

Class identifier of `mxArray`

## Fortran Syntax

```
integer*4 function mxGetClassID(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to an `mxArray`.

## Returns

A numeric identifier that represents the class (category) of the `mxArray` that `pm` points to.

## Description

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

## See Also

`mxGetClassName (Fortran)`

## Purpose

Class of mxArray as string

## C Syntax

```
#include "matrix.h"
const char *mxGetClassName(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

The class (as a string) of `array_ptr`.

## Description

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 `array_ptr` points to a logical `mxArray`, then `mxGetClassName` returns `logical`.

`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.

## Examples

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.

## See Also

`mxGetClassID (C)`

# mxGetClassName (Fortran)

## Purpose

mxArray class as character array

## Fortran Syntax

```
character*(*) function mxGetClassName(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

The class (as a character array) of mxArray, pm.

## Description

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, then mxGetClassName returns logical.

## See Also

mxGetClassID (Fortran)

## Purpose

Pointer to data

## C Syntax

```
#include "matrix.h"
void *mxGetData(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Description

Similar to `mxGetPr`, except `mxGetData` returns a `void *`.

## Examples

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

For additional examples, see `mxcreatecharmatrixfromstr.c` and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxGetImagData (C)`, `mxGetPr (C)`

# mxGetData (Fortran)

## Purpose

Pointer to data

## Fortran Syntax

```
MWPOINTER function mxGetData(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to an `mxArray`.

## Returns

The address of the first element of the real data, on success. Returns `0` if there is no real data or if there is an error.

## Description

Call `mxGetData` to get a pointer to the real data in the `mxArray` that `pm` points to. To copy values from the pointer to Fortran, use one of the `mxCopyPtrTo*` functions in the manner shown here.

```
C      Get the data in mxArray, pm
       mxCopyPtrToReal8(mxGetData(pm), data,
     +                      mxGetNumberOfElements(pm))
```

`mxGetData` is equivalent to using `mxGetPr (Fortran)`.

## See Also

`mxGetImagData (Fortran)`, `mxSetData (Fortran)`, `mxSetImagData (Fortran)`, `mxCopyPtrToReal4 (Fortran)`, `mxCopyPtrToReal8 (Fortran)`, `mxGetPr (Fortran)`

## Purpose

Pointer to dimensions array

## C Syntax

```
#include "matrix.h"
const int *mxGetDimensions(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

The address of the first element in a dimension 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 `array_ptr` points to. Call `mxGetNumberOfDimensions` to get the number of dimensions in the `mxArray`.

## Examples

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

For additional examples, see `findnz.c` and `phonebook.c` in the `refbook` subdirectory of the `examples` directory; see `explore.c` in the `mex` subdirectory of the `examples` directory; see `mxgeteps.c` and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxGetNumberOfDimensions (C)`

# mxGetDimensions (Fortran)

## Purpose

Pointer to dimensions array

## Fortran Syntax

```
MWPOINTER function mxGetDimensions(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to an `mxArray`.

## Returns

A pointer to the first element in a dimension array. Each integer in the dimensions array represents the number of elements in a particular dimension.

## 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`.

`mxGetDimensions` returns a pointer to the dimension array. To copy the values to Fortran, use `mxCopyPtrToInteger4 (Fortran)` in the manner shown here.

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

## See Also

`mxGetNumberOfDimensions (Fortran)`

## Purpose

Number of bytes required to store each data element

## C Syntax

```
#include "matrix.h"
int mxGetElementSize(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
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 `array_ptr` points to an `mxArray` having an unrecognized class. If `array_ptr` points to a cell `mxArray` or a structure `mxArray`, then `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 `mxClassID` of an `mxArray` is `mxINT16_CLASS`, then 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, `memcpy` requires (for its third argument) the size of the elements you intend to copy.

## Examples

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

## See Also

`mxGetM (C)`, `mxGetN (C)`

# mxGetElementSize (Fortran)

## Purpose

Number of bytes required to store each data element

## Fortran Syntax

```
integer*4 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`, then `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 class of an `mxArray` is `int16`, then the `mxArray` stores each data element as a 16-bit (2 byte) signed integer. Thus, `mxGetElementSize` returns `2`.

## See Also

`mxGetM (Fortran)`, `mxGetN (Fortran)`

## Purpose

Value of eps

## C Syntax

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

## Returns

The value of the MATLAB eps variable.

## Description

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.

## Examples

See mxgeteps.c in the mx subdirectory of the examples directory.

## See Also

mxGetInf (C), mxGetNaN (C)

# mxGetEps (Fortran)

## Purpose

Value of `eps`

## Fortran Syntax

```
real*8 function mxGetEps
```

## Returns

The value of the MATLAB `eps` variable.

## Description

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.

## See Also

`mxGetInf (Fortran)`, `mxGetNaN (Fortran)`

## Purpose

Field value, given field name and index into structure array

## C Syntax

```
#include "matrix.h"
mxArray *mxGetField(const mxArray *array_ptr, int index,
          const char *field_name);
```

## Arguments

`array_ptr`
: Pointer to a structure `mxArray`.

`index`
: The desired element. 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`.

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

## Returns

A pointer to the `mxArray` in the specified field at the specified `field_name`, on success. Returns `NULL` if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying an `array_ptr` that does not point to a structure `mxArray`. To determine if `array_ptr` points to a structure `mxArray`, call `mxIsStruct`.
- Specifying an out-of-range `index` to an element past the end of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than 9.
- Specifying a nonexistent `field_name`. 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

```
array_ptr[index].field_name
```

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

---

Calling

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

is equivalent to calling

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

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

## See Also

`mxGetFieldByNumber (C)`, `mxGetFieldNameByNumber (C)`, `mxGetFieldNumber (C)`, `mxGetNumberOfFields (C)`, `mxIsStruct (C)`, `mxSetField (C)`, `mxSetFieldByNumber (C)`

## Purpose

Structure array field value, given field name and index

## Fortran Syntax

```
MWPOINTER function mxGetField(pm, index, fieldname)
MWPOINTER pm
integer*4 index
character*(*) fieldname
```

## Arguments

`pm`
: Pointer to a structure `mxArray`.

`index`
: The desired element. 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 zero if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying a `pm` that does not point to a structure `mxArray`. To determine if `pm` points to a structure `mxArray`, call `mxIsStruct`.
- Specifying an out-of-range `index` to an element past the end of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than 10.
- Specifying a nonexistent `fieldname`. Call `mxGetFieldNameByNumber` 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.

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

---

Calling

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

is equivalent to calling

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

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

## See Also

mxGetFieldByNumber (Fortran), mxGetFieldNameByNumber (Fortran), mxGetNumberOfFields (Fortran), mxIsStruct (Fortran), mxSetField (Fortran), mxSetFieldByNumber (Fortran)

## Purpose

Field value, given field number and index into structure array

## C Syntax

```
#include "matrix.h"
mxArray *mxGetFieldByNumber(const mxArray *array_ptr, int index,
          int field_number);
```

## Arguments

`array_ptr`
: Pointer to a structure `mxArray`.

index
: The desired element. 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`. See `mxCalcSingleSubscript` for more details on calculating an `index`.

`field_number`
: The position of the field whose value you want to extract. 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.

## Returns

A pointer to the `mxArray` in the specified field for the desired element, on success. Returns `NULL` if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying an `array_ptr` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.
- Specifying an index < 0 or >= the number of elements in the array.
- 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 `field_number` 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.

Calling

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

is equivalent to calling

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

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

## Examples

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

For additional examples, see `mxisclass.c` in the `mx` subdirectory of the `examples` directory and `explore.c` in the `mex` subdirectory of the `examples` directory.

## See Also

`mxGetField (C)`, `mxGetFieldNameByNumber (C)`, `mxGetFieldNumber (C)`, `mxGetNumberOfFields (C)`, `mxSetField (C)`, `mxSetFieldByNumber (C)`

## Purpose

Structure array field value, given field number and index

## Fortran Syntax

```
MWPOINTER function mxGetFieldByNumber(pm, index, fieldnumber)
MWPOINTER pm
integer*4 index, fieldnumber
```

## Arguments

`pm`
: Pointer to a structure `mxArray`.

`index`
: The desired element. 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`.

`fieldnumber`
: The position of the field whose value you want to extract. 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.

## Returns

A pointer to the `mxArray` in the specified field for the desired element, on success. Returns zero if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying a `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to is a structure `mxArray`.
- Specifying an index < `1` or > the number of elements in the array.
- Specifying a nonexistent field number. Call `mxGetFieldNumber (Fortran)` 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.

Calling

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

is equivalent to calling

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

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

## See Also

`mxGetField (Fortran)`, `mxGetFieldNameByNumber (Fortran)`, `mxGetNumberOfFields (Fortran)`, `mxSetField (Fortran)`, `mxSetFieldByNumber (Fortran)`

## Purpose

Field name, given field number in structure array

## C Syntax

```
#include "matrix.h"
const char *mxGetFieldNameByNumber(const mxArray *array_ptr,
            int field_number);
```

## Arguments

`array_ptr`
Pointer to a structure `mxArray`.

`field_number`
The position of the desired field. For instance, to get the name of the first field, set `field_number` to 0; to get the name of the second field, set `field_number` to 1; and so on.

## Returns

A pointer to the nth field name, on success. Returns `NULL` on failure. Common causes of failure include

- Specifying an `array_ptr` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.
- Specifying a value of `field_number` greater than or equal to the number of fields in the structure `mxArray`. (Remember that `field_number` 0 symbolizes the first field, so `index N-1` symbolizes the last field.)

## Description

Call `mxGetFieldNameByNumber` to get the name of a field in the given structure `mxArray`. A typical use of `mxGetFieldNameByNumber` is to call 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];
```

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`.

## Examples

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

For additional examples, see `mxisclass.c` in the `mx` subdirectory of the `examples` directory and `explore.c` in the `mex` subdirectory of the `examples` directory.

## See Also

`mxGetField (C)`, `mxIsStruct (C)`, `mxSetField (C)`

## Purpose

Structure array field name, given field number

## Fortran Syntax

```
character*(*) function mxGetFieldNameByNumber(pm, fieldnumber)
MWPOINTER pm
integer*4 fieldnumber
```

## Arguments

`pm`

Pointer to a structure `mxArray`.

`fieldnumber`

The position of the desired field. For instance, to get the name of the first field, set `fieldnumber` to `1`; to get the name of the second field, set `fieldnumber` to `2`; and so on.

## Returns

The nth field name, on success. Returns `0` on failure. Common causes of failure include

- Specifying a `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to a structure `mxArray`.
- Specifying a value of `fieldnumber` greater than the number of fields in the structure `mxArray`. (Remember that `fieldnumber 1` represents the first field, so `index N` represents the last field.)

## Description

Call `mxGetFieldNameByNumber` to get the name of a field in the given structure `mxArray`. A typical use of `mxGetFieldNameByNumber` is to call it inside a loop 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];
```

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`.

## See Also

mxGetField (Fortran), mxIsStruct (Fortran), mxSetField (Fortran)

## Purpose

Field number, given field name in structure array

## C Syntax

```
#include "matrix.h"
int mxGetFieldNumber(const mxArray *array_ptr,
    const char *field_name);
```

## Arguments

`array_ptr`
Pointer to a structure `mxArray`.

`field_name`
The name of a field in the structure `mxArray`.

## Returns

The field number of the specified `field_name`, on success. The first field has a field number of 0, the second field has a field number of 1, and so on. Returns `-1` on failure. Common causes of failure include

- Specifying an `array_ptr` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.
- Specifying the `field_name` of a nonexistent field.

## Description

If you know the name of a field but do not know its field number, call `mxGetFieldNumber`. 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];
```

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 `mxGetFieldNumber` and specify a field name of anything other than `name`, `billing`, or `test`, then `mxGetFieldNumber` returns `-1`.

Calling

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

is equivalent to calling

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

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

## Examples

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

## See Also

`mxGetField (C)`, `mxGetFieldByNumber (C)`, `mxGetFieldNameByNumber (C)`, `mxGetNumberOfFields (C)`, `mxSetField (C)`, `mxSetFieldByNumber (C)`

## Purpose

Structure array field number, given field name

## Fortran Syntax

```
integer*4 function mxGetFieldNumber(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. The first field has a field number of `1`, the second field has a field number of `2`, and so on. Returns `0` on failure. Common causes of failure include

- Specifying a `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `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 `mxGetFieldNumber`. 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];
```

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 `mxGetFieldNumber` and specify a field name of anything other than `'name'`, `'billing'`, or `'test'`, then `mxGetFieldNumber` returns `0`.

Calling

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

is equivalent to calling

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

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

## See Also

`mxGetField (Fortran)`, `mxGetFieldByNumber (Fortran)`, `mxGetFieldNameByNumber (Fortran)`, `mxGetNumberOfFields (Fortran)`, `mxSetField (Fortran)`, `mxSetFieldByNumber (Fortran)`

## Purpose

Pointer to imaginary data of mxArray

## C Syntax

```
#include "matrix.h"
void *mxGetImagData(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Description

Similar to mxGetPi, except it returns a void *.

## Examples

See mxisfinite.c in the mx subdirectory of the examples directory.

## See Also

mxGetData (C), mxGetPi (C)

# mxGetImagData (Fortran)

## Purpose

Pointer to imaginary data of mxArray

## Fortran Syntax

```
MWPOINTER function mxGetImagData(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

The address of the first element of the imaginary data, on success. Returns 0 if there is no imaginary data or if there is an error.

## Description

Call mxGetImagData to determine the starting address of the imaginary data in the mxArray that pm points to. To copy values from the pointer to Fortran, use one of the mxCopyPtrToComplex* functions in the manner shown here.

```
C      Get the real and imaginary data in mxArray, pm
       mxCopyPtrToComplex16(mxGetData(pm), mxGetImagData(pm),
     +               data, mxGetNumberOfElements(pm))
```

mxGetImagData is equivalent to using mxGetPi (Fortran).

## See Also

mxGetData (Fortran), mxSetImagData (Fortran), mxSetData (Fortran), mxCopyPtrToComplex8 (Fortran), mxCopyPtrToComplex16 (Fortran), mxGetPi (Fortran)

## Purpose

Value of infinity

## C Syntax

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

## Returns

The value of infinity on your system.

## Description

Call `mxGetInf` to return the value of the MATLAB internal `inf` variable. `inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `inf` is built into the system; you cannot modify it.

Operations that return infinity include

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

## Examples

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

## See Also

`mxGetEps (C)`, `mxGetNaN (C)`

# mxGetInf (Fortran)

## Purpose

Value of infinity

## Fortran Syntax

```
real*8 function mxGetInf
```

## Returns

The value of infinity on your system.

## Description

Call `mxGetInf` to return the value of the MATLAB internal `inf` variable. `inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `inf` is built into the system. You cannot modify it.

Operations that return infinity include

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

## See Also

`mxGetEps (Fortran)`, `mxGetNaN (Fortran)`

## Purpose

ir array of sparse matrix

## C Syntax

```
#include "matrix.h"
int *mxGetIr(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to a sparse mxArray.

## Returns

A pointer to the first element in the ir array, if successful, and NULL otherwise. Possible causes of failure include

- Specifying a full (nonsparse) mxArray.
- Specifying a NULL array_ptr. (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, then 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.

## Examples

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

For additional examples, see explore.c in the mex subdirectory of the examples directory; see mxsetdimensions.c and mxsetnzmax.c in the mx subdirectory of the examples directory.

## See Also

mxGetJc (C), mxGetNzmax (C), mxSetIr (C), mxSetJc (C), mxSetNzmax (C)

# mxGetIr (Fortran)

## Purpose

ir array

## 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 zero otherwise. Possible causes of failure include

- Specifying a full (nonsparse) mxArray.
- 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, then 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 (Fortran) and mxSetJc (Fortran).

## See Also

mxGetJc (Fortran), mxGetNzmax (Fortran), mxSetIr (Fortran), mxSetJc (Fortran), mxSetNzmax (Fortran)

## Purpose

jc array of sparse matrix

## C Syntax

```
#include "matrix.h"
int *mxGetJc(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to a sparse mxArray.

## Returns

A pointer to the first element in the jc array, if successful, and NULL otherwise. The most likely cause of failure is specifying an array_ptr that points to a full (nonsparse) mxArray.

## Description

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.

## Examples

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

For additional examples, see explore.c in the mex subdirectory of the examples directory; see mxgetnzmax.c, mxsetdimensions.c, and mxsetnzmax.c in the mx subdirectory of the examples directory.

## See Also

mxGetIr (C), mxSetIr (C), mxSetJc (C)

# mxGetJc (Fortran)

## Purpose

jc array

## Fortran Syntax

```
MWPOINTER function mxGetJc(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to a sparse mxArray.

## Returns

A pointer to the first element in the jc array if successful, and zero otherwise. The most likely cause of failure is specifying a pointer that points to a full (nonsparse) mxArray.

## Description

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 (Fortran).

## See Also

mxGetIr (Fortran), mxSetIr (Fortran), mxSetJc (Fortran)

## Purpose

Pointer to logical array data

## C Syntax

```
#include "matrix.h"
mxLogical *mxGetLogicals(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

The address of the first logical in the `mxArray`. Returns `NULL` if the specified array is not a logical array.

## Description

Call `mxGetLogicals` to determine the address of the first logical element in the `mxArray` that `array_ptr` points to. Once you have the starting address, you can access any other element in the `mxArray`.

## See Also

`mxIsLogical (C)`, `mxIsLogicalScalar (C)`, `mxIsLogicalScalarTrue (C)`, `mxCreateLogicalScalar (C)`, `mxCreateLogicalMatrix (C)`, `mxCreateLogicalArray (C)`

# mxGetM (C)

## Purpose

Number of rows in mxArray

## C Syntax

```
#include "matrix.h"
int mxGetM(const mxArray *array_ptr);
```

## Arguments

array_ptr
: Pointer to an array.

## Returns

The number of rows in the mxArray to which array_ptr points.

## Description

mxGetM returns the number of rows in the specified array. The term *rows* always means the first dimension of the array no matter how many dimensions the array has. For example, if array_ptr points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, then mxGetM returns 8.

## Examples

See convec.c in the refbook subdirectory of the examples directory.

For additional examples, see fulltosparse.c, revord.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory; see mxmalloc.c and mxsetdimensions.c in the mx subdirectory of the examples directory; see mexget.c, mexlock.c, mexsettrapflag.c, and yprime.c in the mex subdirectory of the examples directory.

## See Also

mxGetN (C), mxSetM (C), mxSetN (C)

## Purpose

Number of rows in mxArray

## Fortran Syntax

```
integer*4 function mxGetM(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

The number of rows in the mxArray to which pm points.

## Description

mxGetM returns the number of rows in the specified array.

## 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

mxGetN (Fortran), mxSetM (Fortran), mxSetN (Fortran)

# mxGetN (C)

## Purpose

Number of columns in mxArray

## C Syntax

```
#include "matrix.h"
int mxGetN(const mxArray *array_ptr);
```

## Arguments

array_ptr
: 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 array_ptr is an N-dimensional mxArray, mxGetN is the product of dimensions 2 through N. For example, if array_ptr points to a four-dimensional mxArray having dimensions 13-by-5-by-4-by-6, then mxGetN returns the value 120 (5x4x6). If the specified mxArray has more than two dimensions and you need to know exactly how many elements are in each dimension, then call mxGetDimensions.

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

## Examples

See convec.c in the refbook subdirectory of the examples directory.

For additional examples,

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

## See Also

mxGetM (C), mxGetNumberOfDimensions (C), mxSetM (C), mxSetN (C)

## Purpose

Number of columns in mxArray

## Fortran Syntax

```
integer*4 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 points to a sparse mxArray, mxGetN still returns the number of columns, not the number of occupied columns.

## 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 (Fortran), mxSetM (Fortran), mxSetN (Fortran)

# mxGetNaN (C)

## Purpose

Value of `NaN` (Not-a-Number)

## C Syntax

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

## 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.

## Examples

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

## See Also

`mxGetEps (C)`, `mxGetInf (C)`

## Purpose

Value of `NaN` (Not-a-Number)

## 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.

## See Also

`mxGetEps (Fortran)`, `mxGetInf (Fortran)`

# mxGetNumberOfDimensions (C)

## Purpose

Number of dimensions in `mxArray`

## C Syntax

```
#include "matrix.h"
int mxGetNumberOfDimensions(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`

## Returns

The number of dimensions in the specified `mxArray`. The returned value is always 2 or greater.

## Description

Use `mxGetNumberOfDimensions` to determine how many dimensions are in the specified array. To determine how many elements are in each dimension, call `mxGetDimensions`.

## Examples

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

For additional examples, see `findnz.c`, `fulltosparse.c`, and `phonebook.c` in the `refbook` subdirectory of the `examples` directory; see `mxcalcsinglesubscript.c`, `mxgeteps.c`, and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxSetM (C)`, `mxSetN (C)`, `mxGetDimensions (C)`

## Purpose

Number of dimensions in `mxArray`

## Fortran Syntax

```
integer*4 function mxGetNumberOfDimensions(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to an `mxArray`.

## Returns

The number of dimensions in the specified `mxArray`. The returned value is always 2 or greater.

## Description

Use `mxGetNumberOfDimensions` to determine how many dimensions are in the specified array. To determine how many elements are in each dimension, call `mxGetDimensions`.

## See Also

`mxSetM (Fortran)`, `mxSetN (Fortran)`, `mxGetDimensions (Fortran)`

# mxGetNumberOfElements (C)

## Purpose

Number of elements in `mxArray`

## C Syntax

```
#include "matrix.h"
int mxGetNumberOfElements(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

Number of elements in the specified `mxArray`.

## Description

`mxGetNumberOfElements` tells you how many elements an array has. For example, if the dimensions of an array are 3-by-5-by-10, then `mxGetNumberOfElements` will return the number 150.

## Examples

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

For additional examples, see `explore.c` in the `mex` subdirectory of the `examples` directory; see `mxcalcsinglesubscript.c`, `mxgeteps.c`, `mxgetinf.c`, `mxisfinite.c`, and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxGetDimensions (C)`, `mxGetM (C)`, `mxGetN (C)`, `mxGetClassID (C)`, `mxGetClassName (C)`

## Purpose

Number of elements in `mxArray`

## Fortran Syntax

```
integer*4 function mxGetNumberOfElements(pm)
MWPOINTER pm
```

## Arguments

`pm`
Pointer to an `mxArray`.

## Returns

Number of elements in the specified `mxArray`.

## Description

`mxGetNumberOfElements` tells you how many elements an `mxArray` has. For example, if the dimensions of an array are 3-by-5-by-10, then `mxGetNumberOfElements` will return the number `150`.

## See Also

`mxGetDimensions (Fortran)`, `mxGetM (Fortran)`, `mxGetN (Fortran)`, `mxGetClassName (Fortran)`

# mxGetNumberOfFields (C)

## Purpose

Number of fields in structure `mxArray`

## C Syntax

```
#include "matrix.h"
int mxGetNumberOfFields(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to a structure `mxArray`.

## Returns

The number of fields, on success. Returns 0 on failure. The most common cause of failure is that `array_ptr` is not a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` is a structure.

## Description

Call `mxGetNumberOfFields` to determine how many fields are in the specified structure `mxArray`.

Once you know the number of fields in a structure, it is easy to loop through every field in order to set or to get field values.

## Examples

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

For additional examples, see `mxisclass.c` in the `mx` subdirectory of the `examples` directory; see `explore.c` in the `mex` subdirectory of the `examples` directory.

## See Also

`mxGetField (C)`, `mxIsStruct (C)`, `mxSetField (C)`

## Purpose

Number of fields in structure `mxArray`

## Fortran Syntax

```
integer*4 function mxGetNumberOfFields(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to a structure `mxArray`.

## Returns

The number of fields, on success. Returns `0` on failure of if no fields exist. The most common cause of failure is that `pm` is not a structure `mxArray`. Call `mxIsStruct` to determine if `pm` is a structure.

## Description

Call `mxGetNumberOfFields` to determine how many fields are in the specified structure `mxArray`.

Once you know the number of fields in a structure, it is easy to loop through every field to set or to get field values.

## See Also

`mxGetField (Fortran)`, `mxIsStruct (Fortran)`, `mxSetField (Fortran)`

# mxGetNzmax (C)

## Purpose

Number of elements in `ir`, `pr`, and `pi` arrays

## C Syntax

```
#include "matrix.h"
int mxGetNzmax(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to a sparse `mxArray`.

## Returns

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 `array_ptr` points to a full (nonsparse) `mxArray`.

## Description

Use `mxGetNzmax` to get the value of the `nzmax` field. The `nzmax` field holds an integer value that signifies the number of elements in the `ir`, `pr`, and, if it exists, the `pi` arrays. The value of `nzmax` is always greater than or equal to the number of nonzero elements in a sparse `mxArray`. In addition, the value of `nzmax` is always less than or equal to the number of rows times the number of columns.

As you adjust the number of nonzero elements in a sparse `mxArray`, MATLAB often adjusts the value of the `nzmax` field. MATLAB adjusts `nzmax` in order to reduce the number of costly reallocations and in order to optimize its use of heap space.

## Examples

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

## See Also

`mxSetNzmax (C)`

## Purpose

Number of elements in `ir`, `pr`, and `pi` arrays

## Fortran Syntax

```
integer*4 function mxGetNzmax(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to a sparse `mxArray`.

## Returns

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 `pm` points to a full (nonsparse) `mxArray`.

## Description

Use `mxGetNzmax` to get the value of the `nzmax` field. The `nzmax` field holds an integer value that signifies the number of elements in the `ir`, `pr`, and, if it exists, the `pi` arrays. The value of `nzmax` is always greater than or equal to the number of nonzero elements in a sparse `mxArray`. In addition, the value of `nzmax` is always less than or equal to the number of rows times the number of columns.

As you adjust the number of nonzero elements in a sparse `mxArray`, MATLAB often adjusts the value of the `nzmax` field. MATLAB adjusts `nzmax` in order to reduce the number of costly reallocations and in order to optimize its use of heap space.

## See Also

`mxSetNzmax (Fortran)`

# mxGetPi (C)

## Purpose

Imaginary data elements in mxArray

## C Syntax

```
#include "matrix.h"
double *mxGetPi(const mxArray *array_ptr);
```

## Arguments

array_ptr
: Pointer to an mxArray.

## Returns

The imaginary data elements of the specified mxArray, on success. Returns NULL if there is no imaginary data or if there is an error.

## Description

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.

The best way to determine if an mxArray is purely real is to call mxIsComplex.

The imaginary parts of all input matrices to a MATLAB function are allocated if any of the input matrices are complex.

## Examples

See convec.c, findnz.c, and fulltosparse.c in the refbook subdirectory of the examples directory.

For additional examples, see explore.c and mexcallmatlab.c in the mex subdirectory of the examples directory; see mxcalcsinglesubscript.c, mxgetinf.c, mxisfinite.c, and mxsetnzmax.c in the mx subdirectory of the examples directory.

## See Also

mxGetPr (C), mxSetPi (C), mxSetPr (C)

## Purpose

Imaginary data elements of mxArray

## Fortran Syntax

```
MWPOINTER function mxGetPi(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

The imaginary data elements of the specified mxArray, on success. Returns 0 if there is no imaginary data or if there is an error.

## Description

Use mxGetPi to determine the starting address of the imaginary data in the mxArray that pm points to.

See the description for mxGetImagData (Fortran), which is an equivalent function to mxGetPi.

## See Also

mxGetPr (Fortran), mxSetPi (Fortran), mxSetPr (Fortran), mxGetImagData (Fortran)

# mxGetPr (C)

## Purpose

Real data elements in mxArray

## C Syntax

```
#include "matrix.h"
double *mxGetPr(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

The address of the first element of the real data. Returns NULL if there is no real data.

## Description

Call mxGetPr to determine the starting address of the real data in the mxArray that array_ptr points to. Once you have the starting address, you can access any other element in the mxArray.

## Examples

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.

## See Also

mxGetPi (C), mxSetPi (C), mxSetPr (C)

## Purpose

Real data elements of mxArray

## Fortran Syntax

```
MWPOINTER function mxGetPr(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

The address of the first element of the real data. Returns 0 if there is no real data.

## Description

Use mxGetPr to determine the starting address of the real data in the mxArray that pm points to.

See the description for mxGetData (Fortran), which is an equivalent function to mxGetPr.

## 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

mxGetPi (Fortran), mxSetPr (Fortran), mxSetPi (Fortran), mxGetData (Fortran)

# mxGetScalar (C)

## Purpose

Real component of first data element in mxArray

## C Syntax

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

## Arguments

array_ptr
: 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 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 array_ptr points to a structure mxArray or a cell mxArray, mxGetScalar returns 0.0.

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

If array_ptr 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 array_ptr points to an mxArray containing only one element (a scalar). However, array_ptr can point to an mxArray containing many elements. If array_ptr points to an mxArray containing multiple elements, mxGetScalar returns the value of the first real element. If array_ptr points to a two-dimensional mxArray, mxGetScalar returns the value of the (1,1) element; if array_ptr points to a three-dimensional mxArray, mxGetScalar returns the value of the (1,1,1) element; and so on.

## Examples

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

For additional examples, see `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory; see `mexget.c`, `mexlock.c` and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory.

## See Also

`mxGetM (C)`, `mxGetN (C)`

# mxGetScalar (Fortran)

## Purpose

Real component of first data element in mxArray

## Fortran Syntax

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

## Arguments

pm

Pointer to an mxArray.

## Returns

The value of the first real (nonimaginary) element of the mxArray. 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.

## See Also

mxGetM (Fortran), mxGetN (Fortran)

## Purpose

Copy string mxArray to C-style string

## C Syntax

```
#include "matrix.h"
int mxGetString(const mxArray *array_ptr, char *buf, int buflen);
```

## Arguments

array_ptr
: Pointer to a string mxArray; that is, a pointer to an mxArray having the mxCHAR_CLASS class.

buf
: The starting location into which the string should be written. mxGetString writes the character data into buf and then terminates the string with a NULL character (in the manner of C strings). buf can point to either dynamic or static memory.

buflen
: Maximum number of characters to read into buf. Typically, you set buflen to 1 plus the number of elements in the string mxArray to which array_ptr points. See the mxGetM and mxGetN reference pages to find out how to get the number of elements.

## Returns

0 on success, and 1 on failure. Possible reasons for failure include

- Specifying an mxArray that is not a string mxArray.
- Specifying buflen with less than the number of characters needed to store the entire mxArray pointed to by array_ptr. If this is the case, 1 is returned and the string is truncated.

## Description

Call mxGetString to copy the character data of a string mxArray into a C-style string. The copied C-style string starts at buf and contains no more than buflen-1 characters. 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.

For additional examples, see `explore.c` in the `mex` subdirectory of the `examples` directory; see `mxmalloc.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxCreateCharArray (C)`, `mxCreateCharMatrixFromStrings (C)`, `mxCreateString (C)`, `mxArrayToString (C)`

## Purpose

Create character array from mxArray

## Fortran Syntax

```
integer*4 function mxGetString(pm, str, strlen)
MWPOINTER pm
character*(*) str
integer*4 strlen
```

## Arguments

pm
: Pointer to an mxArray.

str
: Fortran character array.

strlen
: Number of characters to retrieve from the mxArray.

## Returns

0 on success, and 1 otherwise.

## Description

Call mxGetString to copy a character array from an mxArray. mxGetString copies and converts the character array from the mxArray pm into the character array str. Storage space for character array str must be allocated previously.

Only up to strlen characters are copied, so ordinarily, strlen is set to the dimension of the character array to prevent writing past the end of the array. Check the length of the character array in advance using mxGetM and mxGetN. If the character array contains several rows, they are copied, one column at a time, into one long character array.

## See Also

mxCalloc (Fortran)

# mxIsCell (C)

## Purpose

Determine whether input is cell `mxArray`

## C Syntax

```
#include "matrix.h"
bool mxIsCell(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an array.

## Returns

Logical `1` (`true`) if `array_ptr` points to an array having the class `mxCELL_CLASS`, and logical `0` (`false`) otherwise.

## Description

Use `mxIsCell` to determine if the specified array is a cell array.

Calling `mxIsCell` is equivalent to calling

```
mxGetClassID(array_ptr) == mxCELL_CLASS
```

---

**Note** `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.

---

## See Also

`mxIsClass (C)`

## Purpose

Determine whether input is cell mxArray

## Fortran Syntax

```
integer*4 function mxIsCell(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an array.

## Returns

Logical 1 (true) if pm points to an array of the MATLAB cell class, and logical 0 (false) otherwise.

## Description

Use mxIsCell to determine if the specified mxArray is a cell array.

Calling mxIsCell is equivalent to calling

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

**Note** 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.

## See Also

mxIsClass (Fortran)

# mxIsChar (C)

## Purpose

Determine whether input is string mxArray

## C Syntax

```
#include "matrix.h"
bool mxIsChar(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

Logical 1 (true) if array_ptr points to an array having the class mxCHAR_CLASS, and logical 0 (false) otherwise.

## Description

Use mxIsChar to determine if array_ptr points to string mxArray.

Calling mxIsChar is equivalent to calling

```
mxGetClassID(array_ptr) == mxCHAR_CLASS
```

## Examples

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.

## See Also

mxIsClass (C), mxGetClassID (C)

## Purpose

Determine whether input is character mxArray

## Fortran Syntax

```
integer*4 function mxIsChar(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

Logical 1 (true) if pm points to an array of the MATLAB char class, and logical 0 (false) otherwise.

## Description

Use mxIsChar to determine if the specified array is a character mxArray.

Calling mxIsChar is equivalent to calling

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

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran)

# mxIsClass (C)

## Purpose

Determine whether `mxArray` is member of specified class

## C Syntax

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

## Arguments

`array_ptr`
Pointer to an array.

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

| Value of Name | Corresponding Class |
|---|---|
| `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` |
| `uint16` | `mxUINT16_CLASS` |
| `uint32` | `mxUINT32_CLASS` |
| `uint64` | `mxUINT64_CLASS` |

| Value of Name | Corresponding Class |
|---|---|
| *<class_name>* | *<class_id>* |
| unknown | mxUNKNOWN_CLASS |

In the table, *<class_name>* represents the name of a specific MATLAB custom object.

Or, you can specify one of your own class names.

For example,

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

is equivalent to calling

```
mxIsDouble(array_ptr);
```

which is equivalent to calling

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

Note that it is most efficient to use the `mxIsDouble` form.

## Returns

Logical `1` (`true`) if `array_ptr` points to an array having category `name`, and logical `0` (`false`) otherwise.

## Description

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

## Examples

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

## See Also

`mxIsEmpty (C)`, `mxGetClassID (C)`, `mxClassID (C)`

# mxIsClass (Fortran)

## Purpose

Determine whether `mxArray` is member of specified class

## Fortran Syntax

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

## Arguments

`pm`

Pointer to an array.

`classname`

A `character` array specifying the class name you are testing for. You can specify any one of the following predefined constants.

| | | | |
|---|---|---|---|
| `cell` | `char` | `double` | `function_handle` |
| `int8` | `int16` | `int32` | `logical` |
| `object` | `single` | `struct` | `uint8` |
| `uint16` | `uint32` | *<class_name>* | `unknown` |

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 if the specified `mxArray` has this type.

## Examples

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

is equivalent to calling either one of the following

```
mxIsDouble(pm)
```

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

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

## See Also

`mxIsEmpty (Fortran)`, `mxGetClassID (Fortran)`

# mxIsComplex (C)

## Purpose

Determine whether data is complex

## C Syntax

```
#include "matrix.h"
bool mxIsComplex(const mxArray *array_ptr);
```

## Returns

Logical `1` (`true`) if `array_ptr` is a numeric array containing complex data, and logical `0` (`false`) otherwise. If `array_ptr` points to a cell array or a structure array, then `mxIsComplex` returns `false`.

## Description

Use `mxIsComplex` to determine whether or not an imaginary part is allocated for an `mxArray`. The imaginary pointer `pi` is `NULL` 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.

## Examples

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

For additional examples, see `convec.c`, `phonebook.c`, `timestwo.c`, and `xtimesy.c` in the `refbook` subdirectory of the `examples` directory; see `explore.c`, `yprime.c`, `mexlock.c`, and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory; see `mxcalcsinglesubscript.c`, `mxgeteps.c`, and `mxgetinf.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxIsNumeric (C)`

## Purpose

Determine whether mxArray is complex

## Fortran Syntax

```
integer*4 function mxIsComplex(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

1 if complex, and 0 otherwise.

## Description

Use mxIsComplex to determine whether or not an imaginary part is allocated for an mxArray. The imaginary pointer pi is 0 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.

## See Also

mxIsNumeric (Fortran)

# mxIsDouble (C)

## Purpose

Determine whether mxArray represents data as double-precision, floating-point numbers

## C Syntax

```
#include "matrix.h"
bool mxIsDouble(const mxArray *array_ptr);
```

## Arguments

array_ptr
: Pointer to an mxArray.

## Returns

Logical 1 (true) if the mxArray stores its data as double-precision, floating-point numbers, and logical 0 (false) otherwise.

## Description

Call mxIsDouble to determine whether or not the specified mxArray represents its real and imaginary data as double-precision, floating-point numbers.

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.

Calling mxIsDouble is equivalent to calling

```
mxGetClassID(array_ptr) == mxDOUBLE_CLASS
```

## Examples

See findnz.c, fulltosparse.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory.

For additional examples, see mexget.c, mexlock.c, mexsettrapflag.c, and yprime.c in the mex subdirectory of the examples directory; see mxcalcsinglesubscript.c, mxgeteps.c, mxgetinf.c, and mxisfinite.c in the mx subdirectory of the examples directory.

## See Also

mxIsClass (C), mxGetClassID (C)

## Purpose

Determine whether mxArray is of type double

## Fortran Syntax

```
integer*4 function mxIsDouble(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

Logical 1 (true) if mxArray is of type double; and logical 0 (false) otherwise. If mxIsDouble returns 0, the array has no Fortran access functions and your Fortran program cannot use it.

## Description

Call mxIsDouble to determine whether or not the specified mxArray represents its real and imaginary data as double-precision, floating-point numbers.

Older versions of MATLAB store all mxArray data as double-precision, floating-point numbers. However, starting with MATLAB 5, MATLAB can store real and imaginary data in a variety of numerical formats.

Calling mxIsDouble is equivalent to calling

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

# mxIsEmpty (C)

## Purpose

Determine whether mxArray is empty

## C Syntax

```
#include "matrix.h"
bool mxIsEmpty(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an array.

## Returns

Logical 1 (true) if the mxArray is empty, and logical 0 (false) otherwise.

## Description

Use mxIsEmpty to determine if an mxArray contains no data. An mxArray is empty if the size of any of its dimensions is 0.

Note that mxIsEmpty is not the opposite of mxIsFull.

## Examples

See mxisfinite.c in the mx subdirectory of the examples directory.

## See Also

mxIsClass (C)

## Purpose

Determine whether mxArray is empty

## Fortran Syntax

```
integer*4 function mxIsEmpty(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an array.

## Returns

Logical 1 (true) if the mxArray is empty, and logical 0 (false) otherwise.

## Description

Use mxIsEmpty to determine if an mxArray contains no data. An mxArray is empty if the size of any of its dimensions is 0.

Note that mxIsEmpty is not the opposite of mxIsFull.

## See Also

mxIsClass (Fortran)

# mxIsFinite (C)

## Purpose

Determine whether input is finite

## C Syntax

```
#include "matrix.h"
bool mxIsFinite(double value);
```

## Arguments

`value`
The double-precision, floating-point number that you are testing.

## Returns

Logical `1` (`true`) if `value` is finite, and logical `0` (`false`) otherwise.

## Description

Call `mxIsFinite` to determine whether or not `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

## Examples

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

## See Also

`mxIsInf (C)`, `mxIsNaN (C)`

## Purpose

Determine whether input is finite

## Fortran Syntax

```
integer*4 function mxIsFinite(value)
real*8 value
```

## Arguments

`value`
The double-precision, floating-point number that you are testing.

## Returns

Logical `1` (`true`) if `value` is finite, and logical `0` (`false`) otherwise.

## Description

Call `mxIsFinite` to determine whether or not `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

## See Also

`mxIsInf (Fortran)`, `mxIsNaN (Fortran)`

# mxIsFromGlobalWS (C)

## Purpose

Determine whether `mxArray` was copied from MATLAB global workspace

## C Syntax

```
#include "matrix.h"
bool mxIsFromGlobalWS(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

Logical `1` (`true`) if the array was copied out of the global workspace, and logical `0` (`false`) otherwise.

## Description

`mxIsFromGlobalWS` is useful for stand-alone MAT programs. `mexIsGlobal` tells you if the pointer you pass actually points into the global workspace.

## Examples

See `matdgns.c` and `matcreat.c` in the `eng_mat` subdirectory of the `examples` directory.

## See Also

`mexIsGlobal (C)`

## Purpose

Determine whether `mxArray` originated from MATLAB global workspace

## Fortran Syntax

```
integer*4 function mxIsFromGlobalWS(pm)
MWPOINTER pm
```

## Arguments

`pm`

Pointer to an `mxArray`.

## Returns

Logical `1` (`true`) if the array originated from the global workspace, and logical `0` (`false`) otherwise.

## Description

Use `mxIsFromGlobalWS` with stand-alone MAT programs to determine if an array was a global variable when it was saved.

## See Also

`mexIsGlobal (Fortran)`

# mxIsInf (C)

## Purpose

Determine whether input is infinite

## C Syntax

```
#include "matrix.h"
bool mxIsInf(double value);
```

## Arguments

`value`
The double-precision, floating-point number that you are testing.

## Returns

Logical `1` (`true`) if value is infinite, and logical `0` (`false`) otherwise.

## Description

Call `mxIsInf` to determine whether or not `value` is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable `Inf` is built into the system; you cannot modify it.

Operations that return infinity include

- Division by 0. For example, `5/0` returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

If `value` equals `NaN` (Not-a-Number), then `mxIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

## Examples

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

## See Also

`mxIsFinite (C)`, `mxIsNaN (C)`

## Purpose

Determine whether input is infinite

## Fortran Syntax

```
integer*4 function mxIsInf(value)
real*8 value
```

## Arguments

`value`

The double-precision, floating-point number that you are testing.

## Returns

Logical `1` (`true`) if value is infinite, and logical `0` (`false`) otherwise.

## Description

Call `mxIsInf` to determine whether or not `value` is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable, `Inf`, is built into the system. You cannot modify it.

Operations that return infinity include

- Division by 0. For example, `5/0` returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

## See Also

`mxIsFinite (Fortran)`, `mxIsNaN (Fortran)`

# mxIsInt16 (C)

## Purpose

Determine whether mxArray represents data as signed 16-bit integers

## C Syntax

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

## Arguments

array_ptr
: 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.

Calling mxIsInt16 is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT16_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsInt8 (C), mxIsInt32 (C), mxIsInt64 (C), mxIsUint8 (C), mxIsUint16 (C), mxIsUint32 (C), mxIsUint64 (C)

## Purpose

Determine whether input is mxArray of signed 16-bit integers

## 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.

Calling mxIsInt16 is equivalent to calling

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

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsInt8 (Fortran), mxIsInt32 (Fortran), mxIsInt64 (Fortran), mxIsUint8 (Fortran), mxIsUint16 (Fortran), mxIsUint32 (Fortran), mxIsUint64 (Fortran)

# mxIsInt32 (C)

## Purpose

Determine whether mxArray represents data as signed 32-bit integers

## C Syntax

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

## Arguments

array_ptr
: 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.

Calling mxIsInt32 is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT32_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsInt8 (C), mxIsInt16 (C), mxIsInt64 (C), mxIsUint8 (C), mxIsUint16 (C), mxIsUint32 (C), mxIsUint64 (C)

## Purpose

Determine whether input is mxArray of signed 32-bit integers

## Fortran Syntax

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

## Arguments

m

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.

Calling mxIsInt32 is equivalent to calling

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

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsInt8 (Fortran), mxIsInt16 (Fortran), mxIsInt64 (Fortran), mxIsUint8 (Fortran), mxIsUint16 (Fortran), mxIsUint32 (Fortran), mxIsUint64 (Fortran)

# mxIsInt64 (C)

## Purpose

Determine whether mxArray represents data as signed 64-bit integers

## C Syntax

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

## Arguments

array_ptr
: 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.

Calling mxIsInt64 is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT64_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsInt8 (C), mxIsInt16 (C), mxIsInt32 (C), mxIsUint8 (C), mxIsUint16 (C), mxIsUint32 (C), mxIsUint64 (C)

## Purpose

Determine whether input is mxArray of signed 64-bit integers

## Fortran Syntax

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

## Arguments

m

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.

Calling mxIsInt64 is equivalent to calling

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

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsInt8 (Fortran), mxIsInt16 (Fortran), mxIsInt32 (Fortran), mxIsUint8 (Fortran), mxIsUint16 (Fortran), mxIsUint32 (Fortran), mxIsUint64 (Fortran)

# mxIsInt8 (C)

## Purpose

Determine whether mxArray represents data as signed 8-bit integers

## C Syntax

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

## Arguments

array_ptr
: 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.

Calling mxIsInt8 is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT8_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsInt16 (C), mxIsInt32 (C), mxIsInt64 (C), mxIsUint8 (C), mxIsUint16 (C), mxIsUint32 (C), mxIsUint64 (C)

## Purpose

Determine whether input is mxArray of signed 8-bit integers

## 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.

Calling mxIsInt8 is equivalent to calling

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

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsInt16 (Fortran), mxIsInt32 (Fortran), mxIsInt64 (Fortran), mxIsUint8 (Fortran), mxIsUint16 (Fortran), mxIsUint32 (Fortran), mxIsUint64 (Fortran)

# mxIsLogical (C)

## Purpose

Determine whether `mxArray` is of class `mxLogical`

## C Syntax

```
#include "matrix.h"
bool mxIsLogical(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

Logical `1` (`true`) if `array_ptr` points to a logical `mxArray`, and logical `0` (`false`) otherwise.

## Description

Use `mxIsLogical` to determine whether MATLAB treats the data in the `mxArray` as Boolean (logical). If an `mxArray` is logical, then 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.

## Examples

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

## See Also

`mxIsClass (C)`

## Purpose

Determine whether mxArray is Boolean

## Fortran Syntax

```
integer*4 function mxIsLogical(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

Logical 1 (true) if pm points to a logical mxArray, and logical 0 (false) otherwise.

## Description

Use mxIsLogical to determine whether MATLAB treats the data in the mxArray as Boolean (logical). If an mxArray is logical, then 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.

## See Also

mxIsClass (Fortran), logical

# mxIsLogicalScalar (C)

## Purpose

Determine whether scalar `mxArray` is of class `mxLogical`

## C Syntax

```
#include "matrix.h"
bool mxIsLogicalScalar(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

Logical `1` (`true`) if the `mxArray` is of class `mxLogical` and has 1-by-1 dimensions, and logical `0` (`false`) otherwise.

## Description

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.

`mxIsLogicalScalar(pa)` is equivalent to

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1
```

## See Also

`mxIsLogicalScalarTrue (C)`, `mxIsLogical (C)`, `mxGetLogicals (C)`, `mxGetScalar (C)`

## Purpose

Determine whether scalar `mxArray` of class `mxLogical` is `true`

## C Syntax

```
#include "matrix.h"
bool mxIsLogicalScalarTrue(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

Logical `1` (`true`) if the value of the `mxArray`'s logical, scalar element is `true`, and logical `0` (`false`) otherwise.

## Description

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.

`mxIsLogicalScalarTrue(pa)` is equivalent to

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1 &&
mxGetLogicals(pa)[0] == true
```

## See Also

`mxIsLogicalScalar (C)`, `mxIsLogical (C)`, `mxGetLogicals (C)`, `mxGetScalar (C)`

# mxIsNaN (C)

## Purpose

Determine whether input is `NaN` (Not-a-Number)

## C Syntax

```
#include "matrix.h"
bool mxIsNaN(double value);
```

## Arguments

`value`
The double-precision, floating-point number that you are testing.

## Returns

Logical `1` (`true`) if `value` is `NaN` (Not-a-Number), and logical `0` (`false`) otherwise.

## Description

Call `mxIsNaN` to determine whether or not `value` 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

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

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.

## Examples

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

For additional examples, see `findnz.c` and `fulltosparse.c` in the `refbook` subdirectory of the `examples` directory.

## See Also

`mxIsFinite (C)`, `mxIsInf (C)`

## Purpose

Determine whether value is NaN (Not-a-Number)

## Fortran Syntax

```
integer*4 function mxIsNaN(value)
real*8 value
```

## Arguments

`value`
The double-precision, floating-point number that you are testing.

## Returns

Logical `1` (`true`) if `value` is `NaN` (Not-a-Number), and logical `0` (`false`) otherwise.

## Description

Call `mxIsNaN` to determine whether or not `value` 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

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

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) uses to represent an error condition or missing data.

## See Also

`mxIsFinite (Fortran)`, `mxIsInf (Fortran)`

# mxIsNumeric (C)

## Purpose

Determine whether `mxArray` is numeric

## C Syntax

```
#include "matrix.h"
bool mxIsNumeric(const mxArray *array_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

## Returns

Logical `1` (`true`) if the array's storage type is

- `mxDOUBLE_CLASS`
- `mxSINGLE_CLASS`
- `mxINT8_CLASS`
- `mxUINT8_CLASS`
- `mxINT16_CLASS`
- `mxUINT16_CLASS`
- `mxINT32_CLASS`
- `mxUINT32_CLASS`
- `mxINT64_CLASS`
- `mxUINT64_CLASS`

Logical `0` (`false`) if the array's storage type is

- `mxCELL_CLASS`
- `mxCHAR_CLASS`
- `mxFUNCTION_CLASS`
- `mxLOGICAL_CLASS`
- `mxSTRUCT_CLASS`
- `mxUNKNOWN_CLASS`

## Description

Call `mxIsNumeric` to determine if the specified array contains numeric data. If the specified array is a cell, string, or a structure, then `mxIsNumeric` returns logical `0` (`false`). Otherwise, `mxIsNumeric` returns logical `1` (`true`).

Call `mxGetClassID` to determine the exact storage type.

## Examples

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

## See Also

`mxGetClassID (C)`

# mxIsNumeric (Fortran)

## Purpose

Determine whether mxArray contains numeric data

## Fortran Syntax

```
integer*4 function mxIsNumeric(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

1 if the mxArray contains numeric data, and 0 otherwise.

## Description

Call mxIsNumeric to inquire whether or not the mxArray contains numeric data, such as data of class double or uint16. Note that logical data is not numeric.

## Examples

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.

## Purpose

Determine whether mxArray represents data as single-precision, floating-point numbers

## C Syntax

```
#include "matrix.h"
bool mxIsSingle(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

Logical 1 (true) if the array stores its data as single-precision, floating-point numbers, and logical 0 (false) otherwise.

## Description

Use mxIsSingle to determine whether or not the specified array represents its real and imaginary data as single-precision, floating-point numbers.

Calling mxIsSingle is equivalent to calling

```
mxGetClassID(array_ptr) == mxSINGLE_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C)

# mxIsSingle (Fortran)

## Purpose

Determine whether input is single-precision, floating-point mxArray

## Fortran Syntax

```
integer*4 function mxIsSingle(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

Logical 1 (true) if the array stores its data as single-precision, floating-point numbers, and logical 0 (false) otherwise.

## Description

Use mxIsSingle to determine whether or not the specified array represents its real and imaginary data as single-precision, floating-point numbers.

Calling mxIsSingle is equivalent to calling

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

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran)

## Purpose

Determine whether input is sparse mxArray

## C Syntax

```
#include "matrix.h"
bool mxIsSparse(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

Logical 1 (true) if array_ptr points to a sparse mxArray, and logical 0 (false) otherwise. A false return value means that array_ptr points to a full mxArray or that array_ptr does not point to a legal mxArray.

## Description

Use mxIsSparse to determine if array_ptr points to a sparse mxArray. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.

## Examples

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.

## See Also

mxGetIr (C), mxGetJc (C)

# mxIsSparse (Fortran)

## Purpose

Determine whether mxArray is sparse

## Fortran Syntax

```
integer*4 function mxIsSparse(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

1 if the mxArray is sparse, and 0 otherwise.

## Description

Use mxIsSparse to determine if an mxArray is stored in sparse form. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.

There are no corresponding set routines. Use mxCreateSparse to create sparse mxArrays.

## See Also

mxGetIr (Fortran), mxGetJc (Fortran), mxCreateSparse (Fortran)

## Purpose

Determine whether input is structure mxArray

## C Syntax

```
#include "matrix.h"
bool mxIsStruct(const mxArray *array_ptr);
```

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

Logical 1 (true) if array_ptr points to a structure mxArray, and logical 0 (false) otherwise.

## Description

Use mxIsStruct to determine if array_ptr points to a structure mxArray. Many routines (for example, mxGetFieldName and mxSetField) require a structure mxArray as an argument.

## Examples

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

## See Also

mxCreateStructArray (C), mxCreateStructMatrix (C), mxGetNumberOfFields (C), mxGetField (C), mxSetField (C)

# mxIsStruct (Fortran)

## Purpose

Determine whether input is structure mxArray

## Fortran Syntax

```
integer*4 function mxIsStruct(pm)
MWPOINTER pm
```

## Arguments

pm

Pointer to an mxArray.

## Returns

Logical 1 (true) if pm points to a structure array; and logical 0 (false) otherwise.

## Description

Use mxIsStruct to determine if pm points to a structure mxArray. Many routines (for example, mxGetFieldName and mxSetField) require a structure mxArray as an argument.

## See Also

mxCreateStructArray (Fortran), mxCreateStructMatrix (Fortran), mxGetNumberOfFields (Fortran), mxGetField (Fortran), mxSetField (Fortran)

## Purpose

Determine whether mxArray represents data as unsigned 16-bit integers

## C Syntax

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

## Arguments

array_ptr
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.

Calling mxIsUint16 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT16_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsUint8 (C), mxIsUint32 (C), mxIsUint64 (C), mxIsInt8 (C), mxIsInt16 (C), mxIsInt32 (C), mxIsInt64 (C)

# mxIsUint16 (Fortran)

## Purpose

Determine whether input is mxArray of unsigned 16-bit integers

## 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.

Calling mxIsUint16 is equivalent to calling

```
mxGetClassName(pm) == 'uint16'
```

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsUint8 (Fortran), mxIsUint32 (Fortran), mxIsUint64 (Fortran), mxIsInt8 (Fortran), mxIsInt16 (Fortran), mxIsInt32 (Fortran), mxIsInt64 (Fortran)

## Purpose

Determine whether mxArray represents data as unsigned 32-bit integers

## C Syntax

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

## Arguments

array_ptr
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.

Calling mxIsUint32 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT32_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsUint8 (C), mxIsUint16 (C), mxIsUint64 (C), mxIsInt8 (C), mxIsInt16 (C), mxIsInt32 (C), mxIsInt64 (C)

# mxIsUint32 (Fortran)

## Purpose

Determine whether input is mxArray of unsigned 32-bit integers

## 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.

Calling mxIsUint32 is equivalent to calling

```
mxGetClassName(pm) == 'uint32'
```

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsUint8 (Fortran), mxIsUint16 (Fortran), mxIsUint64 (Fortran), mxIsInt8 (Fortran), mxIsInt16 (Fortran), mxIsInt32 (Fortran), mxIsInt64 (Fortran)

## Purpose

Determine whether mxArray represents data as unsigned 64-bit integers

## C Syntax

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

## Arguments

array_ptr
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.

Calling mxIsUint64 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT64_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsUint8 (C), mxIsUint16 (C), mxIsUint32 (C), mxIsInt8 (C), mxIsInt16 (C), mxIsInt32 (C), mxIsInt64 (C)

# mxIsUint64 (Fortran)

## Purpose

Determine whether input is mxArray of unsigned 64-bit integers

## 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.

Calling mxIsUint64 is equivalent to calling

```
mxGetClassName(pm) == 'uint64'
```

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsUint8 (Fortran), mxIsUint16 (Fortran), mxIsUint32 (Fortran), mxIsInt8 (Fortran), mxIsInt16 (Fortran), mxIsInt32 (Fortran), mxIsInt64 (Fortran)

## Purpose

Determine whether mxArray represents data as unsigned 8-bit integers

## C Syntax

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

## Arguments

array_ptr
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.

Calling mxIsUint8 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT8_CLASS
```

## See Also

mxIsClass (C), mxGetClassID (C), mxIsUint16 (C), mxIsUint32 (C), mxIsUint64 (C), mxIsInt8 (C), mxIsInt16 (C), mxIsInt32 (C), mxIsInt64 (C)

# mxIsUint8 (Fortran)

## Purpose

Determine whether input is mxArray of unsigned 8-bit integers

## Fortran Syntax

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

## Arguments

`m`

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.

Calling mxIsUint8 is equivalent to calling

```
mxGetClassName(pm) == 'uint8'
```

## See Also

mxIsClass (Fortran), mxGetClassID (Fortran), mxIsUint16 (Fortran), mxIsUint32 (Fortran), mxIsUint64 (Fortran), mxIsInt8 (Fortran), mxIsInt16 (Fortran), mxIsInt32 (Fortran), mxIsInt64 (Fortran)

## Purpose

Allocate dynamic memory using MATLAB memory manager

## C Syntax

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

## Arguments

`n`

Number of bytes to allocate.

## Returns

A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxMalloc` returns `NULL`. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.

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

## Description

MATLAB applications should always call `mxMalloc` rather than `malloc` to allocate memory. Note that `mxMalloc` works differently in MEX-files than in stand-alone MATLAB applications.

In MEX-files, `mxMalloc` automatically

- Allocates enough contiguous heap space to hold `n` bytes.
- Registers the returned heap space with the MATLAB memory management facility.

The MATLAB memory management facility maintains a list of all memory allocated by `mxMalloc`. 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 applications, `mxMalloc` calls the ANSI C `malloc` function.

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.

## 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

mxCalloc (C), mxRealloc (C), mxFree (C), mxDestroyArray (C), mexMakeArrayPersistent (C), mexMakeMemoryPersistent (C)

## Purpose

Allocate dynamic memory using MATLAB memory manager

## Fortran Syntax

```
MWPOINTER function mxMalloc(n)
integer*4 n
```

## Arguments

`n`

Number of bytes to allocate.

## Returns

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

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

## Description

Use `mxMalloc` to allocate dynamic memory using the MATLAB memory management facility.

MATLAB maintains a list of all memory allocated by `mxMalloc`. MATLAB automatically frees (deallocates) all of a MEX-file's memory when the MEX-file completes and control returns to the MATLAB prompt.

If you want the memory to persist after a 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.

Note that `mxMalloc` works differently in MEX-files than in stand-alone MATLAB applications.

In MEX-files, `mxMalloc` automatically

- Allocates enough contiguous heap space to hold `n` bytes.
- Registers the returned heap space with the MATLAB memory management facility.

## See Also

mxCalloc (Fortran), mxRealloc (Fortran), mxFree (Fortran), mxDestroyArray (Fortran), mexAtExit (Fortran), mexMakeArrayPersistent (Fortran), mexMakeMemoryPersistent (Fortran)

## Purpose

Reallocate memory

## C Syntax

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

## Arguments

ptr

Pointer to a block of memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

`size`

New size of allocated memory, in bytes.

## Returns

A pointer to the reallocated block of memory, or `NULL` 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 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.

## Description

`mxRealloc` changes the size of a memory block that has been allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc`.

If `size` is 0 and `ptr` is not `NULL`, `mxRealloc` frees the memory pointed to by `ptr` and returns `NULL`.

If `size` is greater than 0 and `ptr` is `NULL`, `mxRealloc` behaves like `mxMalloc`, allocating a new block of memory of `size` bytes and returning a pointer to the new block.

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 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`

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.

## Examples

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

## See Also

`mxCalloc (C)`, `mxFree (C)`, `mxMalloc (C)`

## Purpose

Reallocate memory

## Fortran Syntax

```
MWPOINTER function mxRealloc(ptr, size)
MWPOINTER ptr
integer*4 size
```

## Arguments

`ptr`
: Pointer to a block of memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

`size`
: New size of allocated memory, in bytes.

## Returns

A pointer to the reallocated block of memory, or `0` 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 0. 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.

## Description

`mxRealloc` changes the size of a memory block that has been allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc`.

If `size` is 0 and `ptr` is not `0`, `mxRealloc` frees the memory pointed to by `ptr` and returns `0`.

If `size` is greater than 0 and `ptr` is `0`, `mxRealloc` behaves like `mxMalloc`, allocating a new block of memory of `size` bytes and returning a pointer to the new block.

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 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 `0` 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.

## See Also

`mxCalloc (Fortran)`, `mxFree (Fortran)`, `mxMalloc (Fortran)`

## Purpose

Remove field from structure array

## C Syntax

```
#include "matrix.h"
extern void mxRemoveField(mxArray array_ptr, int field_number);
```

## Arguments

`array_ptr`
: Pointer to a structure `mxArray`.

`field_number`
: The number of the field you want to remove. For instance, to remove the first field, set `field_number` to 0; to remove the second field, set `field_number` to 1; and so on.

## Description

Call `mxRemoveField` 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 `mxDestroyArray` to destroy the actual field values.

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];
```

The field number 0 represents the field `name`; field number 1 represents field `billing`; field number 2 represents field `test`.

## See Also

`mxAddField (C)`, `mxDestroyArray (C)`, `mxGetFieldByNumber (C)`

# mxRemoveField (Fortran)

## Purpose

Remove field from structure `mxArray`

## Fortran Syntax

```
subroutine mxRemoveField(pm, fieldnumber)
MWPOINTER pm
integer*4 fieldnumber
```

## Arguments

`pm`

Pointer to a structure `mxArray`.

`fieldnumber`

The number of the field you want to remove. For instance, to remove the first field, set `fieldnumber` to `1`; to remove the second field, set `fieldnumber` to `2`; and so on.

## Description

Call `mxRemoveField` 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 `mxDestroyArray` to destroy the actual field values.

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];
```

The field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`.

## See Also

`mxAddField (Fortran)`, `mxDestroyArray (Fortran)`, `mxGetFieldByNumber (Fortran)`

## Purpose

Set value of one cell of mxArray

## C Syntax

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

## Arguments

`array_ptr`
: 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.

## 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 (C), mxCreateCellMatrix (C), mxGetCell (C), mxIsCell (C), mxFree (C)

## Purpose

Set value of one cell of cell mxArray

## Fortran Syntax

```
subroutine mxSetCell(pm, index, value)
MWPOINTER pm, value
integer*4 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 the index in a multidimensional cell array is to call mxCalcSingleSubscript (Fortran).

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. Use one of the mxCreate* functions to create the value mxArray.

## 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.

## See Also

mxCreateCellArray (Fortran), mxCreateCellMatrix (Fortran), mxGetCell (Fortran), mxIsCell (Fortran), mxFree (Fortran)

# mxSetClassName (C)

## Purpose

Convert structure array to MATLAB object array

## C Syntax

```
#include "matrix.h"
int mxSetClassName(mxArray *array_ptr, const char *classname);
```

## Arguments

`array_ptr`
Pointer to an `mxArray` of class `mxSTRUCT_CLASS`.

`classname`
The object class to which to convert `array_ptr`.

## Returns

0 if successful, and nonzero otherwise.

## Description

`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.

## See Also

`mxIsClass (C)`, `mxGetClassID (C)`

## Purpose

Set pointer to data

## C Syntax

```
#include "matrix.h"
void mxSetData(mxArray *array_ptr, void *data_ptr);
```

## Arguments

`array_ptr`
Pointer to an `mxArray`.

`data_ptr`
Pointer to data.

## Description

`mxSetData` is similar to `mxSetPr`, except its `data_ptr` argument is a `void *`. Use this on numeric arrays with contents other than `double`.

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`.

## See Also

`mxSetPr (C)`, `mxGetData (C)`, `mxFree (C)`

# mxSetData (Fortran)

## Purpose

Set pointer to data

## Fortran Syntax

```
subroutine mxSetData(pm, pr)
MWPOINTER pm, pr
```

## Arguments

`pm`

Pointer to an `mxArray`.

`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 `mxCalloc (Fortran)` to allocate this dynamic memory.

## Description

Use `mxSetData` to set the real data of the specified `mxArray`.

All `mxCreate*` calls allocate heap space to hold real data. Therefore, you do not ordinarily use `mxSetData` to initialize the real elements of a freshly created `mxArray`. Rather, you call `mxSetData` to replace the initial real values with new ones.

`mxSetData` is equivalent to using `mxSetPr (Fortran)`.

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`.

## See Also

`mxSetImagData (Fortran)`, `mxGetData (Fortran)`, `mxGetImagData (Fortran)`, `mxSetPr (Fortran)`, `mxFree (Fortran)`

## Purpose

Modify number of dimensions and size of each dimension

## C Syntax

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

## Arguments

`array_ptr`
: 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, setting `dims[0]` to 5 and `dims[1]` 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`, then 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`, then 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.

## Examples

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

## See Also

`mxGetNumberOfDimensions (C)`, `mxSetM (C)`, `mxSetN (C)`, `mxRealloc (C)`

## Purpose

Modify number of dimensions and size of each dimension

## Fortran Syntax

```
integer*4 function mxSetDimensions(pm, dims, ndim)
MWPOINTER pm
integer*4 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, 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` array. Consequently, if your call to `mxSetDimensions` increases the number of elements in the `mxArray`, then you must enlarge the `pr` (and `pi`, if it exists) array accordingly.

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

## See Also

mxGetNumberOfDimensions (Fortran), mxSetM (Fortran), mxSetN (Fortran)

## Purpose

Set structure array field, given field name and index

## C Syntax

```
#include "matrix.h"
void mxSetField(mxArray *array_ptr, int index,
  const char *field_name, mxArray *value);
```

## Arguments

`array_ptr`
: Pointer to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.

`index`
: The desired element. 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`. See `mxCalcSingleSubscript` for details on calculating an index.

`field_name`
: 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

```
array_ptr[index].field_name = value;
```

---

**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.

---

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);
```

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`.

## Examples

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

## See Also

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

## Purpose

Set structure array field value, given field name and index

## Fortran Syntax

```
subroutine mxSetField(pm, index, fieldname, value)
MWPOINTER pm, value
integer*4 index
character*(*) fieldname
```

## Arguments

`pm`

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

`index`

The desired element. 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 are assigning. Call `mxGetFieldNameByNumber` to determine existing field names.

`value`

Pointer to the `mxArray` you are assigning. Use one of the `mxCreate*` functions to create the `value` `mxArray`.

## Description

Use `mxSetField` to assign a `value` to the specified element of the specified field. `mxSetField` is almost identical to `mxSetFieldByNumber (Fortran)`; however, the former takes a field name as its third argument, and the latter takes a field number 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.

---

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`.

## See Also

`mxCreateStructArray (Fortran)`, `mxCreateStructMatrix (Fortran)`, `mxGetField (Fortran)`, `mxGetFieldByNumber (Fortran)`, `mxGetFieldNameByNumber (Fortran)`, `mxGetNumberOfFields (Fortran)`, `mxIsStruct (Fortran)`, `mxSetFieldByNumber (Fortran)`, `mxFree (Fortran)`

## Purpose

Set structure array field, given field number and index

## C Syntax

```
#include "matrix.h"
void mxSetFieldByNumber(mxArray *array_ptr, int index,
  int field_number, mxArray *value);
```

## Arguments

`array_ptr`
: Pointer to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.

`index`
: The desired element. 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`. See `mxCalcSingleSubscript` for details on calculating an index.

`field_number`
: The position of the field whose value you want to extract. 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.

`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 `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.

---

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);
```

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`.

## 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.

## See Also

`mxCreateStructArray (C)`, `mxCreateStructMatrix (C)`, `mxGetField (C)`, `mxGetFieldByNumber (C)`, `mxGetFieldNameByNumber (C)`, `mxGetFieldNumber (C)`, `mxGetNumberOfFields (C)`, `mxIsStruct (C)`, `mxSetField (C)`, `mxFree (C)`

## Purpose

Set structure array field value, given field number and index

## Fortran Syntax

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

## Arguments

`pm`

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

`index`

The desired element. 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`.

`fieldnumber`

The position of the field whose value you want to extract. 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`, where `N` is the number of fields.

`value`

The value you are assigning. Use one of the `mxCreate*` functions to create the `value` `mxArray`.

## Description

Use `mxSetFieldByNumber` to assign a `value` to the specified element of the specified field. `mxSetFieldByNumber` is almost identical to `mxSetField (Fortran)`; 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.

---

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`.

## See Also

`mxCreateStructArray (Fortran)`, `mxCreateStructMatrix (Fortran)`, `mxGetField (Fortran)`, `mxGetFieldByNumber (Fortran)`, `mxGetFieldNameByNumber (Fortran)`, `mxGetNumberOfFields (Fortran)`, `mxIsStruct (Fortran)`, `mxSetField (Fortran)`, `mxFree (Fortran)`

## Purpose

Set imaginary data pointer for mxArray

## C Syntax

```
#include "matrix.h"
void mxSetImagData(mxArray *array_ptr, void *pi);
```

## Arguments

`array_ptr`
Pointer to an mxArray.

`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 errors will result when the array is destroyed.

## Description

mxSetImagData is similar to mxSetPi (C), except its pi argument is a void *. Use this on numeric arrays with contents other than double.

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.

## Examples

See mxisfinite.c in the mx subdirectory of the examples directory.

## See Also

mxSetPi (C), mxGetImagData (C), mxFree (C)

# mxSetImagData (Fortran)

## Purpose

Set imaginary data pointer for mxArray

## Fortran Syntax

```
subroutine mxSetImagData(pm, pi)
MWPOINTER pm, pi
```

## Arguments

pm

Pointer to an mxArray.

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 (Fortran) to allocate this dynamic memory. If pi points to static memory, memory errors will result when the array is destroyed.

## Description

Use mxSetImagData to set the imaginary data of the specified mxArray.

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 COMPLEX = 1 or by setting pi to a nonzero value), then you do not ordinarily use mxSetImagData to initialize the created mxArray's imaginary elements. Rather, you call mxSetImagData to replace the initial imaginary values with new ones.

mxSetImagData is equivalent to using mxSetPi (Fortran).

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.

## See Also

mxSetData (Fortran), mxGetImagData (Fortran), mxGetData (Fortran), mxSetPi (Fortran), mxFree (Fortran)

## Purpose

Set ir array of sparse mxArray

## C Syntax

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

## Arguments

array_ptr
: 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, then 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.

## 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.

## See Also

mxCreateSparse (C), mxGetIr (C), mxGetJc (C), mxSetJc (C), mxFree (C)

## Purpose

Set ir array of sparse mxArray

## 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.)

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.

## See Also

mxCreateSparse (Fortran), mxGetIr (Fortran), mxGetJc (Fortran), mxSetJc (Fortran), mxFree (Fortran)

# mxSetJc (C)

## Purpose

Set jc array of sparse mxArray

## C Syntax

```
#include "matrix.h"
void mxSetJc(mxArray *array_ptr, int *jc);
```

## Arguments

array_ptr
: Pointer to a sparse mxArray.

jc
: Pointer to the jc array.

## Description

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.

If the jth column of the sparse mxArray has any nonzero elements:

- jc[j] is the index in ir, pr, and pi (if it exists) of the first nonzero element in the jth column.
- jc[j+1]-1 is the index of the last nonzero element in the jth column.

The number of nonzero elements in the jth column of the sparse mxArray is

```
jc[j+1] - jc[j];
```

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.

For example, consider a 7-by-3 sparse mxArray named Sparrow containing six nonzero elements, created 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 contents of the `ir`, `jc`, and `pr` arrays are

| Subscript | ir | pr | jc | Comment |
|---|---|---|---|---|
| (2,1) | 1 | 1 | 0 | Column 1 contains two nonzero elements, with rows designated by `ir[0]` and `ir[1]` |
| (5,1) | 4 | 1 | 2 | Column 2 contains one nonzero element, with row designated by `ir[2]` |
| (3,2) | 2 | 1 | 3 | Column 3 contains three nonzero elements, with rows designated by `ir[3]`,`ir[4]`, and `ir[5]` |
| (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

| 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]` |

| Subscript | ir | pr | jc | Comment |
|---|---|---|---|---|
| (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`.

## 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

`mxGetIr (C)`, `mxGetJc (C)`, `mxSetIr (C)`, `mxFree (C)`

## Purpose

Set jc array of sparse mxArray

## Fortran Syntax

```
subroutine mxSetJc(pm, jc)
MWPOINTER pm, jc
```

## Arguments

pm
: Pointer to a sparse mxArray.

jc
: Pointer to the jc array.

## Description

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.

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.

## See Also

mxGetIr (Fortran), mxGetJc (Fortran), mxSetIr (Fortran), mxFree (Fortran)

# mxSetM (C)

## Purpose

Set number of rows in mxArray

## C Syntax

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

## Arguments

`m`
The desired number of rows.

`array_ptr`
Pointer to an mxArray.

## 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, then 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, then 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.

## 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 (C), mxGetN (C), mxSetN (C)

## Purpose

Set number of rows of mxArray

## Fortran Syntax

```
subroutine mxSetM(pm, m)
MWPOINTER pm
integer*4 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. Call mxSetN to set the number of columns.

You can 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, then you must enlarge the pr, pi, ir, and/or jc arrays.

If your calls to mxSetM and mxSetN end up reducing the number of elements in the array, then you may want to reduce the sizes of the pr, pi, ir, and/or jc arrays in order to use heap space more efficiently.

## See Also

mxGetM (Fortran), mxGetN (Fortran), mxSetN (Fortran)

# mxSetN (C)

## Purpose

Set number of columns in mxArray

## C Syntax

```
#include "matrix.h"
void mxSetN(mxArray *array_ptr, int n);
```

## Arguments

`array_ptr`
Pointer to an mxArray.

`n`
The desired number of columns.

## Description

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 array_ptr points to an mxArray having three dimensions, calling mxSetN reduces the mxArray to two dimensions.

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, then you must enlarge the pr, pi, ir, and/or jc arrays.

If your calls to mxSetM and mxSetN end up reducing the number of elements in the mxArray, then 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.

## 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 (C), mxGetN (C), mxSetM (C)

## Purpose

Set number of columns of mxArray

## Fortran Syntax

```
subroutine mxSetN(pm, n)
MWPOINTER pm
integer*4 n
```

## Arguments

pm
: Pointer to an mxArray.

n
: The desired number of columns.

## Description

Call mxSetN to set the number of columns in the specified mxArray. Call mxSetM to set the number of rows in the specified mxArray.

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, then you must enlarge the pr, pi, ir, and/or jc arrays.

If your calls to mxSetM and mxSetN end up reducing the number of elements in the mxArray, then 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.

## See Also

mxGetM (Fortran), mxGetN (Fortran), mxSetM (Fortran)

# mxSetNzmax (C)

## Purpose

Set storage space for nonzero elements

## C Syntax

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

## Arguments

`array_ptr`
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 `mxCalloc`, setting `n` to the new value of `nzmax`.
2. Call the ANSI C routine `memcpy` to copy the contents of the old array to the new area allocated in Step 1.
3. Call `mxFree` to free the memory occupied by the old array.
4. 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.

## Examples

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

## See Also

`mxGetNzmax (C)`

# mxSetNzmax (Fortran)

## Purpose

Set storage space for nonzero elements

## Fortran Syntax

```
subroutine mxSetNzmax(pm, nzmax)
MWPOINTER pm
integer*4 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.

How big should `nzmax` be? One thought is that you set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space. Another technique is to make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

## See Also

`mxGetNzmax (Fortran)`

## Purpose

Set new imaginary data for mxArray

## C Syntax

```
#include "matrix.h"
void mxSetPi(mxArray *array_ptr, double *pi);
```

## Arguments

array_ptr
: Pointer to a full (nonsparse) mxArray.

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.

## Description

Use mxSetPi to set the imaginary data of the specified mxArray.

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 or by setting pi to a non-NULL value), then 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.

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.

## Examples

See mxisfinite.c and mxsetnzmax.c in the mx subdirectory of the examples directory.

## See Also

mxSetImagData (C), mxGetPi (C), mxGetPr (C), mxSetPr (C), mxFree (C)

# mxSetPi (Fortran)

## Purpose

Set new imaginary data for mxArray

## Fortran Syntax

```
subroutine mxSetPi(pm, pi)
MWPOINTER pm, pi
```

## Arguments

pm

Pointer to a full (nonsparse) mxArray.

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 (Fortran) to allocate this dynamic memory. If pi points to static memory, memory errors will result when the array is destroyed.

## Description

Use mxSetPi to set the imaginary data of the specified mxArray.

See the description for mxSetImagData (Fortran), which is an equivalent function to mxSetPi.

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.

## See Also

mxSetPr (Fortran), mxGetPi (Fortran), mxGetPr (Fortran), mxSetImagData (Fortran), mxFree (Fortran)

## Purpose

Set new real data for mxArray

## C Syntax

```
#include "matrix.h"
void mxSetPr(mxArray *array_ptr, double *pr);
```

## Arguments

array_ptr
: Pointer to a full (nonsparse) mxArray.

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 mxCalloc to allocate this dynamic memory. If pr points to static memory, then memory leaks and other memory errors can result.

## Description

Use mxSetPr to set the real data of the specified mxArray.

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.

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.

## Examples

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

## See Also

mxGetPr (C), mxGetPi (C), mxSetPi (C), mxFree (C)

# mxSetPr (Fortran)

## Purpose

Set new real data for mxArray

## Fortran Syntax

```
subroutine mxSetPr(pm, pr)
MWPOINTER pm, pr
```

## Arguments

pm

Pointer to a full (nonsparse) mxArray.

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 mxCalloc (Fortran) to allocate this dynamic memory.

## Description

Use mxSetPr to set the real data of the specified mxArray.

See the description for mxSetData (Fortran), which is an equivalent function to mxSetPr.

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.

## See Also

mxSetPi (Fortran), mxGetPr (Fortran), mxGetPi (Fortran), mxSetData (Fortran), mxFree (Fortran)

# Index

## S