# MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

Application Program Interface Reference

*Version 6*

**How to Contact The MathWorks:**


| | | |
|---|---|---|
| ☎ | 508-647-7000 | Phone |
| | 508-647-7001 | Fax |
| ✉ | The MathWorks, Inc.<br>3 Apple Hill Drive<br>Natick, MA 01760-2098 | Mail |
| | `http://www.mathworks.com` | Web |
| | `ftp.mathworks.com` | Anonymous FTP server |
| | `comp.soft-sys.matlab` | Newsgroup |
| @ | `support@mathworks.com` | Technical support |
| | `suggest@mathworks.com` | Product enhancement suggestions |
| | `bugs@mathworks.com` | Bug reports |
| | `doc@mathworks.com` | Documentation error reports |
| | `subscribe@mathworks.com` | Subscribing user registration |
| | `service@mathworks.com` | Order status, license renewals, passcodes |
| | `info@mathworks.com` | Sales, pricing, and general information |


*Application Program Interface Reference*


© COPYRIGHT 1984 - 2000 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and Target Language Compiler is a trademark of The MathWorks, Inc.

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



| | | |
|---|---|---|
| Printing History: | December 1996 | First printing |
| | May 1997 | Revised for 5.1 (online version) |
| | January 1998 | Revised for 5.2 (online version) |
| | January 1999 | Revised for 5.3 (online version) |
| | September 2000 | Revised for 6.0 (online version) |

# Contents

## 1 API Notes



## 2 C Engine Routines



## 3 C MAT-File Routines

# 4 C MEX-Functions

## 5 C MX-Functions

## 6 Fortran Engine Routines



## 7 Fortran MAT-File Routines



## 8 Fortran MEX-Functions

# 9 Fortran MX-Functions

## 10 DDE Functions

# 1

# API Notes

| | |
|---|---|
| The mex Script | Compiles MEX-function from C or Fortran source code |
| The MATLAB Array | Information on object type used by the MATALB language |
| Passing Pointers in Fortran | Implications of using pointers in Fortran |

**Purpose**

Compiles a MEX-function from C or Fortran source code

**Syntax**

```
MEX <options> <files>
```

**Arguments**

All nonsource code filenames passed as arguments are passed to the linker without being compiled.

These options are available on all platforms except where noted.

| Option | Function |
|---|---|
| `@<rsp_file>` | Include the contents of the text file `<rsp_file>` as command line arguments to the `mex` script. |
| `-argcheck` | Perform argument checking on MATLAB API functions (C functions only). |
| `-c` | Compile only. Do not link. |
| `-D<name>[#<def>]` | Define C preprocessor macro `<name>` [as having value `<def>`]. (Note: UNIX also allows `-D<name>[=<def>]`.) |
| `-f <file>` | Use `<file>` as the options file; `<file>` is a full pathname if it is not in current directory. |
| `-g` | Build an executable with debugging symbols included. |
| `-h[elp]` | Help. Lists the switches and their functions. |
| `-I<pathname>` | Include `<pathname>` in the compiler `include` search path. |
| `-inline` | Inlines matrix accessor functions (`mx*`). The generated MEX-function may not be compatible with future versions of MATLAB. |
| `-l<file>` | For UNIX, link against library `lib<file>`. |
| `-L<pathname>` | For UNIX, include `<pathname>` in the list of directories to search for libraries. |

| Option | Function |
|---|---|
| <name>#<def> | Override options file setting for variable <name>. This option is equivalent to <ENV_VAR>#<val>, which temporarily sets the environment variable <ENV_VAR> to <val> for the duration of the call to mex. <val> can refer to another environment variable by prepending the name of the variable with a $, e.g., COMPFLAGS#"$COMPFLAGS -myswitch". |
| <name>=<def> | For UNIX, override options file setting for variable <name>. |
| -O | Build an optimized executable. |
| -outdir <name> | Place all output files in directory <name>. |
| -output <name> | Create an executable named <name>. An appropriate executable extension is automatically appended. |
| -setup | For Windows, set up default options file. This switch should be the only argument passed. |
| -U<name> | Undefine C preprocessor macro <name>. |
| -v | Verbose. Print all compiler and linker settings. |
| -V4 | Compile MATLAB 4-compatible MEX-file. |

## Description

MEX <options> <files> compiles a MEX-function from C or Fortran source code. All nonsource code filenames passed as arguments are passed to the linker without being compiled.

MEX's execution is affected by both command-line arguments and an options file. The options file contains all compiler-specific information necessary to create a MEX-function. The default name for this options file, if none is specified with the -f option, is mexopts.bat (Windows) and mexopts.sh (UNIX).

**Note** The MathWorks provides an option (setup) for the mex script that lets you set up a default options file on your system.

On UNIX, the options file is written in the Bourne shell script language. The mex script searches for the first occurrence of the options file called mexopts.sh in the following list:

- The current directory
- $HOME/matlab
- <MATLAB>/bin

mex uses the first occurrence of the options file it finds. If no options file is found, mex displays an error message. You can directly specify the name of the options file using the -f switch.

Any variable specified in the options file can be overridden at the command line by use of the <name>=<def> command-line argument. If <def> has spaces in it, then it should be wrapped in single quotes (e.g., OPTFLAGS='opt1 opt2'). The definition can rely on other variables defined in the options file; in this case the variable referenced should have a prepended $ (e.g., OPTFLAGS='$OPTFLAGS opt2').

On Windows, the options file is written in the Perl script language. The default options file is placed in your user profile directory after you configure your system by running mex -setup. The mex script searches for the first occurrence of the options file called mexopts.bat in the following list:

- The current directory
- The user profile directory
- <MATLAB>\bin\win32\mexopts

mex uses the first occurrence of the options file it finds. If no options file is found, mex searches your machine for a supported C compiler and uses the factory default options file for that compiler. If multiple compilers are found, you are prompted to select one.

No arguments can have an embedded equal sign (=); thus, -DFOO is valid, but -DFOO=BAR is not.

# The MATLAB Array

The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, structures, and objects are stored as MATLAB arrays. In C, the MATLAB array is declared to be of type `mxArray`. The `mxArray` structure contains, among other things:

- Its type
- Its dimensions
- The data associated with this array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and fieldnames

## Data Storage

All MATLAB data is stored columnwise. This is how Fortran stores matrices; MATLAB uses this convention because it was originally written in Fortran. For example, given the matrix

```
a=['house'; 'floor'; 'porch']

a =
house
floor
porch
```

Its dimensions are

```
size(a)

ans =
3    5
```

and its data is stored as

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| h | f | p | o | l | o | u | o | r | s | o | c | e | r | h |

## Data Types in MATLAB

### Complex Double-Precision Matrices

The most common data type in MATLAB is the complex double-precision, nonsparse matrix. These matrices are of type `double` and have dimensions `m`-by-`n`, where `m` is the number of rows and `n` is the number of columns. The data is stored as two vectors of double-precision numbers — one contains the real data and one contains the imaginary data. The pointers to this data are referred to as `pr` (pointer to real data) and `pi` (pointer to imaginary data), respectively. A real-only, double-precision matrix is one whose `pi` is `NULL`.

### Numeric Matrices

MATLAB also supports other types of numeric matrices. These are single-precision floating-point and 8-, 16-, and 32-bit integers, both signed and unsigned. The data is stored in two vectors in the same manner as double-precision matrices.

### MATLAB Strings

MATLAB strings are of type `char` and are stored the same way as unsigned 16-bit integers except there is no imaginary data component. Each character in the string is stored as 16-bit ASCII Unicode. Unlike C, MATLAB strings are not null terminated.

### Sparse Matrices

Sparse matrices have a different storage convention than full matrices in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but there are three additional parameters, `nzmax`, `ir`, and `jc`:

- `nzmax` is an integer that contains the length of `ir`, `pr`, and, if it exists, `pi`. It is the maximum possible number of nonzero elements in the sparse matrix.
- `ir` points to an integer array of length `nzmax` containing the row indices of the corresponding elements in `pr` and `pi`.
- `jc` points to an integer array of length `N+1` that contains column index information. For `j`, in the range `0 ≤ j ≤ N-1`, `jc[j]` is the index in `ir` and `pr` (and `pi` if it exists) of the first nonzero entry in the `j`th column and `jc[j+1] - 1` index of the last nonzero entry. As a result, `jc[N]` is also equal to `nnz`, the number of nonzero entries in the matrix. If `nnz` is less than `nzmax`, then

more nonzero entries can be inserted in the array without allocating additional storage.

### Cell Arrays

Cell arrays are a collection of MATLAB arrays where each `mxArray` is referred to as a cell. This allows MATLAB arrays of different types to be stored together. Cell arrays are stored in a similar manner to numeric matrices, except the data portion contains a single vector of pointers to `mxArrays`. Members of this vector are called cells. Each cell can be of any supported data type, even another cell array.

### Structures

A 1-by-1 structure is stored in the same manner as a 1-by-n cell array where `n` is the number of fields in the structure. Members of the data vector are called fields. Each field is associated with a name stored in the `mxArray`.

### Objects

Objects are stored and accessed the same way as structures. In MATLAB, objects are named structures with registered methods. Outside MATLAB, an object is a structure that contains storage for an additional classname that identifies the name of the object.

### Multidimensional Arrays

MATLAB arrays of any type can be multidimensional. A vector of integers is stored where each element is the size of the corresponding dimension. The storage of the data is the same as matrices.

### Logical Arrays

Any noncomplex numeric or sparse array can be flagged as logical. The storage for a logical array is the same as the storage for a nonlogical array.

### Empty Arrays

MATLAB arrays of any type can be empty. An empty `mxArray` is one with at least one dimension equal to zero. For example, a double-precision `mxArray` of type `double`, where `m` and `n` equal `0` and `pr` is `NULL`, is an empty array.

The MATLAB API works with a unique data type, the `mxArray`. Because there is no way to create a new data type in Fortran, MATLAB passes a special identifier, called a pointer, to a Fortran program. You can get information about an `mxArray` by passing this pointer to various API functions called *access routines*. These access routines allow you to get a native Fortran data type containing exactly the information you want, i.e., the size of the `mxArray`, whether or not it is a string, or its data contents.

There are several implications when using pointers in Fortran:

**1** The `%VAL` construct

If your Fortran compiler supports the `%VAL` construct, then there is one type of pointer you can use without requiring an access routine, namely a pointer to data (i.e., the pointer returned by `mxGetPr` or `mxGetPi`). You can use `%VAL` to pass this pointer's contents to a subroutine, where it is declared as a Fortran double-precision array.

If your Fortran compiler does not support the `%VAL` construct, you must use the `mxCopy__` routines (e.g., `mxCopyPtrToReal8`) to access the contents of the pointer.

**2** Variable declarations

To use pointers properly, you must declare them to be the correct size. On DEC Alpha machines, all pointers should be declared as `integer*8`. On all other platforms, pointers should be declared as `integer*4`.

If your Fortran compiler supports preprocessing with the C preprocessor, you can use the preprocessing stage to map pointers to the appropriate declaration. In UNIX, see the examples ending with `.F` in the `examples` directory for a possible approach.

---

**Note** Declaring a pointer to be the incorrect size can cause your program to crash.

---

2

# C Engine Routines

| | |
|---|---|
| `engClose` | Quit engine session |
| `engEvalString` | Evaluate expression in string |
| `engGetArray` | Copy variable from engine workspace |
| `engGetFull (Obsolete)` | Use `engGetArray` followed by appropriate `mxGet` routines |
| `engGetMatrix (Obsolete)` | Use `engGetArray` |
| `engOpen` | Start engine session |
| `engOpenSingleUse` | Start engine session for single, nonshared use |
| `engOutputBuffer` | Specify buffer for MATLAB output |
| `engPutArray` | Put variables into engine workspace |
| `engPutFull (Obsolete)` | Use `mxCreateDoubleMatrix` and `engPutArray` |
| `engPutMatrix (Obsolete)` | Use `engPutArray` |
| `engSetEvalCallback (Obsolete)` | Do not use in programs that interface with MATLAB 5 or later |
| `engSetEvalTimeout (Obsolete)` | Do not use in programs that interface with MATLAB 5 or later |
| `engWinInit (Obsolete)` | Do not use in programs that interface with MATLAB 5 or later |

# engClose

**Purpose**

Quit a 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** 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 MATLAB's *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 via ActiveX.

**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.

# engGetArray

## Purpose

Copy a variable from a MATLAB engine's workspace

## C Syntax

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

## Arguments

`ep`
Engine pointer.

`name`
Name of `mxArray` to get from engine.

## Description

`engGetArray` reads the named `mxArray` from the engine pointed to by `ep` and returns a pointer to a newly allocated `mxArray` structure, or `NULL` if the attempt fails. `engGetArray` will fail if:

- The named variable does not exist.
- In V4-compatible mode if the named variable is not a MATLAB 4 data type.

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

## Example

**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

`engPutArray`

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

`engGetArray` followed by appropriate `mxGet` routines (`mxGetM`, `mxGetN`, `mxGetPr`, `mxGetPi`)

instead of

`engGetFull`

For example,

```
int engGetFull(
   Engine      *ep,    /* engine pointer */
   char        *name,  /* full array name */
   int         *m,     /* returned number of rows */
   int         *n,     /* returned number of columns */
   double      **pr,   /* returned pointer to real part */
   double      **pi    /* returned pointer to imaginary part */
   )
{
   mxArray     *pmat;

   pmat = engGetArray(ep, name);

   if (!pmat)
           return(1);

   if (!mxIsDouble(pmat)) {
           mxDestroyArray(pmat);
           return(1);
   }

   *m  = mxGetM(pmat);
   *n  = mxGetN(pmat);
   *pr = mxGetPr(pmat);
   *pi = mxGetPi(pmat);
```

```
    /* Set pr & pi in array struct to NULL so it can be cleared. */
    mxSetPr(pmat, NULL);
    mxSetPi(pmat, NULL);

    mxDestroyArray(pmat);

    return(0);
}
```

**See Also**

`engGetArray` and examples in the `eng_mat` subdirectory of the `examples` directory

## V4 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
engGetArray
```

instead of

```
engGetMatrix
```

## See Also

`engGetArray`, `engPutArray`, and examples in the `eng_mat` subdirectory of the `examples` directory

# engOpen

**Purpose** Start a 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 an ActiveX 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 ActiveX 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.

# engOpenSingleUse

## Purpose

Start a 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 an ActiveX 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 an ActiveX MATLAB engine server. `engOpenSingleUse` differs from `engOpen`, which allows multiple users to use the same ActiveX MATLAB engine server.

See Introducing MATLAB ActiveX 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.

`n`
Length of buffer `p`.

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

**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`.

**Example**

**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.

# engPutArray

**Purpose** Put variables into a MATLAB engine's workspace

**C Syntax**

```
#include "engine.h"
int engPutArray(Engine *ep, const mxArray *mp);
```

**Arguments**

`ep`
Engine pointer.

`mp`
`mxArray` pointer.

**Description**

`engPutArray` writes `mxArraymp` 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`.

`engPutArray` returns 0 if successful and 1 if an error occurs. In V4 compatibility mode, `engPutArray` will fail if the `mxArray mp` is not a MATLAB 4 data type.

**Example**

**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.

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

`mxCreateDoubleMatrix` and `engPutArray`

instead of

`engPutFull`

For example,

```
int engPutFull(
   Engine      *ep,        /* engine pointer */
   char        *name,      /* full array name */
   int         m,          /* number of rows */
   int         n,          /* number of columns */
   double      *pr,        /* pointer to real part */
   double      *pi         /* pointer to imaginary part */
   )
{
   mxArray     *pmat;
   int         retval;

   pmat = mxCreateDoubleMatrix(0, 0, mxCOMPLEX);

   mxSetName(pmat, name);
   mxSetM(pmat, m);
   mxSetN(pmat, n);
   mxSetPr(pmat, pr);
   mxSetPi(pmat, pi);

   retval = engPutArray(ep, pmat);

   /* Set pr & pi in array struct to NULL so it can be cleared. */
   mxSetPr(pmat, NULL);
   mxSetPi(pmat, NULL);

   mxDestroyArray(pmat);
```

```
    return(retval);
}
```

**See Also** engGetArray, mxCreateDoubleMatrix

## V4 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
engPutArray
```

instead of

```
engPutMatrix
```

## See Also

`engPutArray`

# engSetEvalCallback (Obsolete)

**V4 Compatible** This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later.

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later.

# engWinInit (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function is not necessary in MATLAB 5 or later engine programs.

3

# C MAT-File Routines

| | |
|---|---|
| `matClose` | Close MAT-file |
| `matDeleteArray` | Delete named `mxArray` from MAT-file |
| `matDeleteMatrix (Obsolete)` | Use `matDeleteArray` |
| `matGetArray` | Read `mxArray` from MAT-file |
| `matGetArrayHeader` | Load header array information only |
| `matGetDir` | Get directory of `mxArrays` in MAT-file |
| `matGetFp` | Get file pointer to MAT-file |
| `matGetFull (Obsolete)` | Use `matGetArray` followed by the appropriate `mxGet` routines |
| `matGetMatrix (Obsolete)` | Use `matGetArray` |
| `matGetNextArray` | Read next `mxArray` from MAT-file |
| `matGetNextArrayHeader` | Load array header information only |
| `matGetNextMatrix (Obsolete)` | Use `matGetNextArray` |
| `matGetString (Obsolete)` | Use `matGetArray` and `mxGetString` |
| `matOpen` | Open MAT-file |
| `matPutArray` | Write `mxArrays` into MAT-files |
| `matPutArrayAsGlobal` | Put `mxArrays` into MAT-files |
| `matPutFull (Obsolete)` | Use `mxCreateDoubleMatrix` and `matPutArray` |
| `matPutMatrix (Obsolete)` | Use `matPutArray` |
| `matPutString (Obsolete)` | Use `mxCreateString` and `matPutArray` |

**Purpose** Closes a 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.

**Example** 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.

# matDeleteArray

**Purpose** Delete named `mxArray` from MAT-file

**C Syntax**

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

**Arguments**

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray` to delete.

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

**Example** 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.

## V4 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
matDeleteArray
```

instead of

```
matDeleteMatrix
```

## See Also

`matDeleteArray`

# matGetArray

**Purpose**

Read mxArrays from MAT-files

**C Syntax**

```
#include "mat.h"
mxArray *matGetArray(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.

`matGetArray` 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.

**Example**

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** Load array header information only

**C Syntax**

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

**Arguments**

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray`.

**Description**

`matGetArrayHeader` 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 non-`NULL` when loaded with `matGetArray`, `matGetArrayHeader` 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.

**Example**

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

# matGetDir

**Purpose**

Get directory of mxArrays in a 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**

Get file pointer to a 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.

**Example**

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.

# matGetFull (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the -V4 option of the mex script.

Use

`matGetArray` followed by the appropriate `mxGet` routines

instead of

`matGetFull`

For example,

```
int matGetFull(MATFile *fp, char *name, int *m, int *n,
               double **pr, double **pi)
{
    mxArray *parr;
    /* Get the matrix. */
    parr = matGetArray(fp, name);

    if (parr == NULL)
       return(1);

    if (!mxIsDouble(parr)) {
       mxDestroyArray(parr);
       return(1);
    }
    /* Set up return args. */

    *m  = mxGetM(parr);
    *n  = mxGetN(parr);
    *pr = mxGetPr(parr);
    *pi = mxGetPi(parr);
    /* Zero out pr & pi in array struct so the mxArray can be
       destroyed. */
    mxSetPr(parr, (void *)0);
    mxSetPi(parr, (void *)0);

    mxDestroyArray(parr);
```

```
    return(0);
}
```

**See Also** matGetArray

# matGetMatrix (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
matGetArray
```

instead of

```
matGetMatrix
```

**See Also**

`matGetArray`

**Purpose** Read next mxArray from MAT-file

**C Syntax**

```
#include "mat.h"
mxArray *matGetNextArray(MATFile *mfp);
```

**Arguments**

mfp
Pointer to MAT-file information.

**Description**

matGetNextArray allows you to step sequentially through a MAT-file and read all the mxArrays in a single pass.

matGetNextArray reads the next mxArray from the MAT-file pointed to by mfp and returns a pointer to a newly allocated mxArray structure. Use it 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.

matGetNextArray 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.

**Example**

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.

# matGetNextArrayHeader

**Purpose**

Load array header information only

**C Syntax**

```
#include "mat.h"
mxArray *matGetNextArrayHeader(MATFile *mfp);
```

**Arguments**

`mfp`
Pointer to MAT-file information.

**Description**

`matGetNextArrayHeader` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`, from the file's current file offset.

**Example**

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**

`matGetNextArray`, `matGetArrayHeader`

## V4 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
matGetNextArray
```

instead of

```
matGetNextMatrix
```

## See Also

`matGetNextArray`

# matGetString (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
#include "mat.h"
#include "matrix.h"
mxArray *matGetArray(MATFile *mfp, const char *name);
int mxGetString(const mxArray *array_ptr, char *buf, int buflen)
```

instead of

```
matGetString
```

**See Also**

`matGetArray`, `mxGetString`

## Purpose

Opens a MAT-file

## C Syntax

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

## Arguments

`filename`
Name of file to open.

`mfp`
Pointer to MAT-file information.

`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 MATLAB 4 MAT-file. |

## 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.

## Example

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.

# matPutArray

## Purpose

Write mxArrays into MAT-files

## C Syntax

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

## Arguments

mfp
Pointer to MAT-file information.

mp
mxArray pointer.

## Description

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

matPutArray 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 than the existing mxArray.

matPutArray 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.

## Example

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.

# matPutArrayAsGlobal

**Purpose**

Put mxArrays into MAT-files

**C Syntax**

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

**Arguments**

mfp
Pointer to MAT-file information.

mp
mxArray pointer.

**Description**

This routine allows you to put an mxArray into a MAT-file. matPutArrayAsGlobal is similar to matPutArray, except the array is loaded by MATLAB into the global workspace and a reference to it is set in the local workspace. If you write to a MATLAB 4 format file, matPutArrayAsGlobal will not load it as global, and will act the same as matPutArray.

matPutArrayAsGlobal 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 than the existing mxArray.

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

**Example**

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.

# matPutFull (Obsolete)

**V4 Compatible** This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the -V4 option of the mex script.

Use

`mxCreateDoubleMatrix` and `matPutArray`

instead of

`matPutFull`

For example,

```
int matPutFull(MATFile*ph, char *name, int m, int n, double *pr,
               double *pi)
{
   int         retval;
   mxArray     *parr;

   /* Get empty array struct to place inputs into. */
   parr = mxCreateDoubleMatrix(0, 0, 0);
   if (parr == NULL)
       return(1);

   /* Place inputs into array struct. */
   mxSetM(parr, m);
   mxSetN(parr, n);
   mxSetName(parr, name);
   mxSetPr(parr, pr);
   mxSetPi(parr, pi);

   /* Use put to place array on file. */
   retval = matPutArray(ph, parr);

   /* Zero out pr & pi in array struct so the mxArray can be
      destroyed. */
   mxSetPr(parr, (void *)0);
   mxSetPi(parr, (void *)0);

   mxDestroyArray(parr);
```

```
    return(retval);
}
```

**See Also** mxCreateDoubleMatrix, matPutArray

# matPutMatrix (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
matPutArray
```

instead of

```
matPutMatrix
```

**See Also**

`matPutArray`

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the -V4 option of the mex script.

Use

```
#include "matrix.h"
#include "mat.h"
mxArray *mxCreateString(char *str)
int matPutArray(MATFile *mfp, const mxArray *mp);
void mxDestroyArray(mxArray *array_ptr)
```

instead of

```
matPutString
```

**See Also**

matPutArray

4

# C MEX-Functions

| | |
|---|---|
| `mexAddFlops (Obsolete)` | Update MATLAB's internal floating-point operations counter |
| `mexAtExit` | Register function to be called when MATLAB is cleared or terminates |
| `mexCallMATLAB` | Call MATLAB function or user-defined M-file or MEX-file |
| `mexErrMsgTxt` | Issue error message and return to MATLAB |
| `mexEvalString` | Execute MATLAB command in caller's workspace |
| `mexFunction` | Entry point to C MEX-file |
| `mexFunctionName` | Name of current MEX-function |
| `mexGet` | Get value of Handle Graphics property |
| `mexGetArray` | Get copy of variable from another workspace |
| `mexGetArrayPtr` | Get read-only pointer to variable from another workspace |
| `mexGetEps (Obsolete)` | Use `mxGetEps` |
| `mexGetFull (Obsolete)` | Use `mexGetArray` and `mxGetName`, `mxGetM`, `mxGetN`, `mxGetPr`, `mxGetPi` |
| `mexGetGlobal (Obsolete)` | Use `mexGetArrayPtr` |
| `mexGetInf (Obsolete)` | Use `mxGetInf` |
| `mexGetMatrix (Obsolete)` | Use `mexGetArray` |
| `mexGetMatrixPtr (Obsolete)` | Use `mexGetArrayPtr` |
| `mexGetNaN (Obsolete)` | Use `mxGetNaN` |
| `mexIsFinite (Obsolete)` | Use `mxIsFinite` |
| `mexIsGlobal` | True if `mxArray` has global scope |
| `mexIsInf (Obsolete)` | Use `mxIsInf` |

| | |
|---|---|
| `mexIsLocked` | True if MEX-file is locked |
| `mexIsNaN (Obsolete)` | Use `mxIsNaN` |
| `mexLock` | Lock MEX-file so it cannot be cleared from memory |
| `mexMakeArrayPersistent` | Make `mxArray` persist after MEX-file completes |
| `mexMakeMemoryPersistent` | Make memory allocated by MATLAB's memory allocation routines persist after MEX-file completes |
| `mexPrintf` | ANSI C `printf`-style output routine |
| `mexPutArray` | Copy `mxArray` from your MEX-file into another workspace |
| `mexPutFull (Obsolete)` | Use `mxCreateDoubleMatrix` and `mxSetName` and `mexPutArray` |
| `mexPutMatrix (Obsolete)` | Use `mexPutArray` |
| `mexSet` | Set value of Handle Graphics property |
| `mexSetTrapFlag` | Control response of `mexCallMATLAB` to errors |
| `mexUnlock` | Unlock MEX-file so it can be cleared from memory |
| `mexWarnMsgTxt` | Issue warning message |

**Compatibility**

This API function is obsolete and should not be used in any MATLAB program. This function will not be available in a future version of MATLAB.

**Purpose**

Register a function to be called when the MEX-file is cleared or when 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-file is cleared or MATLAB is terminated. `mexAtExit` gives your MEX-file 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-file 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-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`.

**Example**

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

**See Also**

`mexLock`, `mexUnlock`

**Purpose**

Call a MATLAB function, or a 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, mexSetTrapFlag

**Purpose**

Issue error message and return to the 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**

`mexWarnMsgTxt`

# mexEvalString

**Purpose** Execute a MATLAB command in the workspace of the caller

**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.

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

**See Also** `mexCallMATLAB`

**Purpose**

Entry point to a 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`.

**Example**

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

**Purpose**

Gives the name of the current MEX-function

**C Syntax**

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

**Arguments**

`none`

**Returns**

The name of the current MEX-function.

**Description**

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

**Example**

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

# mexGet

**Purpose**

Get the value of the 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 MATLAB's `get` function. To set a graphics property value, call `mexSet`.

**Example**

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

**See Also**

`mexSet`

**Purpose**

Get a copy of a variable from another workspace

**C Syntax**

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

**Arguments**

`name`
Name of the variable to copy into the MEX-file workspace.

`workspace`
Specifies where `mexGetArray` should search in order to find variable `name`. The possible values are:

| | |
|---|---|
| `base` | Search for variable `name` in the current MATLAB workspace. |
| `caller` | Search for variable `name` in the workspace of whatever entity (M-file, another MEX-file, MATLAB) called this MEX-file. |
| `global` | Search for variable `name` in the list of global variables. If variable `name` exists but is not tagged as a global variable, then `mexGetArray` returns `NULL`. |

**Returns**

A copy of the `mxArray` on success. Returns `NULL` on failure. A common cause of failure is specifying a `name` not currently in the workspace. Perhaps the variable was in the workspace at one time but has since been cleared.

**Description**

Call `mexGetArray` to copy the specified variable name into your MEX-file's workspace. Once inside your MEX-file's workspace, your MEX-file may examine or modify the variable's data and characteristics.

The returned `mxArray` contains a copy of all the data and characteristics that variable `name` had in the other workspace. `mexGetArray` initializes the `name` field of the returned `mxArray` to the variable `name`.

**Example**

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

**See Also**

`mexGetArrayPtr`, `mexPutArray`

# mexGetArrayPtr

**Purpose** Get a read-only pointer to a variable from another workspace

**C Syntax**

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

**Arguments**

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

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

| | |
|---|---|
| `base` | Search the current variables of MATLAB. |
| `caller` | Search the current variables of whatever entity (M-file, another MEX-file, MATLAB workspace) called this MEX-file. |
| `global` | Search the current global variables of MATLAB only. |

**Returns** A read-only pointer `mxArray` called `name` on success. Returns `NULL` on failure.

**Description** Call `mexGetArrayPtr` to get a read-only copy of the specified variable `name` into your MEX-file's workspace. This command is useful for examining an `mxArray's` data and characteristics, but useless for changing them. If you need to change data or characteristics, call `mexGetArray` instead of `mexGetArrayPtr`. If you simply need to examine data or characteristics, `mexGetArrayPtr` offers superior performance as the caller need pass only a pointer to the array. By contrast, `mexGetArray` passes back the entire array.

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

**See Also** `mexGetArray`

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
eps = mxGetEps();
```

instead of

```
eps = mexGetEps();
```

**See Also**

`mxGetEps`

# mexGetFull (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mexGetArray(array_ptr, "caller");
name = mxGetName(array_ptr);
m = mxGetM(array_ptr);
n = mxGetN(array_ptr);
pr = mxGetPr(array_ptr);
pi = mxGetPi(array_ptr);
```

instead of

```
mexGetFull(name, m, n, pr, pi);
```

**See Also**

`mexGetArray`, `mxGetName`, `mxGetPr`, `mxGetPi`

## V4 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mexGetArrayPtr(name, "global");
```

instead of

```
mexGetGlobal(name);
```

## See Also

`mexGetArray`, `mxGetName`, `mxGetPr`, `mxGetPi`

# mexGetInf (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
inf = mxGetInf();
```

instead of

```
inf = mexGetInf();
```

**See Also**

`mxGetInf`

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mexGetArray(name, "caller");
```

instead of

```
mexGetMatrix(name);
```

**See Also**

`mexGetArray`

# mexGetMatrixPtr (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mexGetArrayPtr(name, "caller");
```

instead of

```
mexGetMatrixPtr(name);
```

**See Also**

`mexGetArrayPtr`

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
NaN = mxGetNaN();
```

instead of

```
NaN = mexGetNaN();
```

**See Also**

`mxGetNaN`

# mexIsFinite (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
answer = mxIsFinite(value);
```

instead of

```
answer = mexIsFinite(value);
```

**See Also**

`mxIsFinite`

**Purpose**

True if mxArray has global scope

**C Syntax**

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

**Arguments**

array_ptr
Pointer to an mxArray.

**Returns**

True if the mxArray has global scope, and false otherwise.

**Description**

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

By default, mxArrays have local scope, meaning that changes made to the mxArray inside a MEX-file or stand-alone application have no effect on a variable of the same name in another workspace. However, if an mxArray has global scope, then changes made to the mxArray inside a MEX-file or stand-alone application can affect other workspaces.

The MATLAB global command gives global scope to a MATLAB variable. For example, to make variable x global, just type

```
global x
```

The most common use of mexIsGlobal is to determine if an mxArray stored inside a MAT-files is global.

**Example**

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

**See Also**

mexGetArray, mexGetArrayPtr, mexPutArray

# mexIsInf (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
answer = mxIsInf(value);
```

instead of

```
answer = mexIsInf(value);
```

**See Also**

`mxIsInf`

**Purpose**

True if this MEX-file is locked

**C Syntax**

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

**Returns**

`True` if the MEX-file is locked; `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 a MEX-file at any time. Calling `mexLock` locks a MEX-file, which makes it impossible for a user to clear a MEX-file.

**Example**

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

**See Also**

`mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

# mexIsNaN (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
answer = mxIsNaN(value);
```

instead of

```
answer = mexIsNaN(value);
```

**See Also**

`mxIsInf`

**Purpose**

Lock a MEX-file so that it cannot be 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.

**Example**

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

**See Also**

`mexIsLocked`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

# mexMakeArrayPersistent

**Purpose**

Make an mxArray persist after the 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. MATLAB's 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 the 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, mexLock, mexMakeMemoryPersistent, and the mxCreate functions.

# mexMakeMemoryPersistent

**Purpose**

Make memory allocated by MATLAB's memory allocation routines (`mxCalloc`, `mxMalloc`, `mxRealloc`) persist after the MEX-file completes

**C Syntax**

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

**Arguments**

`ptr`
Pointer to the beginning of memory allocated by one of MATLAB's 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` to see how to register a function that gets called when the MEX-file is cleared. See `mexLock` to see how to lock your MEX-file so that it is never cleared.

---

**See Also**

`mexAtExit`, `mexLock`, `mexMakeArrayPersistent`, `mxCalloc`, `mxFree`, `mxMalloc`, `mxRealloc`

# mexPrintf

**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.

**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`, `mexWarnMsgTxt`

**Purpose** Copy an mxArray from your MEX-file into another workspace

**C Syntax**

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

**Arguments**

array_ptr
Pointer to an mxArray.

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

| | |
|---|---|
| base | Copy name to the current MATLAB workspace. |
| caller | Copy name to the workspace of whatever entity (M-file, another MEX-file, MATLAB workspace) actually called this MEX-file. |
| global | Copy name to the list of global variables. |

**Returns** 0 on success; 1 on failure. A possible cause of failure is that array_ptr is NULL. Another possibility is that array_ptr points to an mxArray that does not have an associated name. (Call mxSetName to associate a name with array_ptr.)

**Description** Call mexPutArray to copy the specified mxArray from your MEX-file into another workspace. mexPutArray makes the specified array accessible to other entities, such as MATLAB, M-files or other MEX-files.

It is easy to confuse array_ptr with a variable name. You manipulate variable names in the MATLAB workspace; you manipulate array_ptrs in a MEX-file. When you call mexPutArray, you specify an array_ptr; however, the recipient workspace appears to receive a variable name. MATLAB determines the variable name by looking at the name field of the received mxArray.

If a variable of the same name already exists in the specified workspace, mexPutArray 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 `mexPutArray` to copy `Peaches` into the MATLAB workspace.

```
mxSetName(array_ptr, "Peaches")
mexPutArray(array_ptr, "base")
```

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

**Example**

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

**See Also**

`mexGetArray`

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
array_ptr = mxCreateDoubleMatrix(0, 0, mxREAL/mxCOMPLEX);
mxSetName(array_ptr, name);
mexPutArray(array_ptr, "caller");
```

instead of

```
mexPutFull(name, m, n, pr, pi)
```

**See Also**

`mxSetM`, `mxSetN`, `mxSetPr`, `mxSetPi`, `mxSetName`, `mexPutArray`

# mexPutMatrix (Obsolete)

**V4 Compatible**

This API function is obsolete and should not be used in a program that interfaces with MATLAB 5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mexPutArray(array_ptr, "caller");
```

instead of

```
mexPutMatrix(matrix_ptr);
```

**See Also**

`mexPutArray`

**Purpose**

Set the value of the 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`
A Handle Graphics property.

`value`
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 MATLAB's `set` function. To get the value of a graphics property, call `mexGet`.

**Example**

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

**See Also**

`mexGet`

# mexSetTrapFlag

**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 MATLAB's 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.

**Example**

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

**See Also**

mexAtExit, mexErrMsgTxt

**Purpose**

Unlock this MEX-file so that it can 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 a MEX-file can be cleared.

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

**Example**

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

**See Also**

`mexIsLocked`, `mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

# mexWarnMsgTxt

## 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 `error_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

`mexErrMsgTxt`

5

# C MX-Functions

| | |
|---|---|
| `mxAddField` | Add field to structure array |
| `mxArrayToString` | Convert arrays to strings |
| `mxAssert` | Check assertion value |
| `mxAssertS` | Check assertion value; doesn't print assertion's text |
| `mxCalcSingleSubscript` | Return offset from first element to desired element |
| `mxCalloc` | Allocate dynamic memory |
| `mxChar` | String `mxArrays` data type |
| `mxClassID` | Enumerated data type that identifies `mxArray's` class |
| `mxClearLogical` | Clear logical flag |
| `mxComplexity` | Specifies if `mxArray` has imaginary components |
| `mxCreateCellArray` | Create unpopulated N-dimensional cell `mxArray` |
| `mxCreateCellMatrix` | Create unpopulated two-dimensional cell `mxArray` |
| `mxCreateCharArray` | Create unpopulated N-dimensional string `mxArray` |
| `mxCreateCharMatrixFromStrings` | Create populated two-dimensional string `mxArray` |
| `mxCreateDoubleMatrix` | Create unpopulated two-dimensional, double-precision, floating-point `mxArray` |
| `mxCreateFull (Obsolete)` | Use `mxCreateDoubleMatrix` |
| `mxCreateNumericArray` | Create unpopulated N-dimensional numeric `mxArray` |
| `mxCreateNumericMatrix` | Create numeric matrix and initialize data elements to 0 |

| | |
|---|---|
| `mxCreateScalarDouble` | Create scalar, double-precision array initialized to specified value |
| `mxCreateSparse` | Create two-dimensional unpopulated sparse `mxArray` |
| `mxCreateString` | Create 1-by-n string `mxArray` initialized to specified string |
| `mxCreateStructArray` | Create unpopulated N-dimensional structure `mxArray` |
| `mxCreateStructMatrix` | Create unpopulated two-dimensional structure `mxArray` |
| `mxDestroyArray` | Free dynamic memory allocated by an `mxCreate` routine |
| `mxDuplicateArray` | Make deep copy of array |
| `mxFree` | Free dynamic memory allocated by `mxCalloc` |
| `mxFreeMatrix (Obsolete)` | Use `mxDestroyArray` |
| `mxGetCell` | Get cell's contents |
| `mxGetClassID` | Get `mxArray`'s class |
| `mxGetClassName` | Get `mxArray`'s class |
| `mxGetData` | Get pointer to data |
| `mxGetDimensions` | Get pointer to dimensions array |
| `mxGetElementSize` | Get number of bytes required to store each data element |
| `mxGetEps` | Get value of `eps` |
| `mxGetField` | Get field value, given field name and index in structure array |
| `mxGetFieldByNumber` | Get field value, given field number and index in structure array |
| `mxGetFieldNameByNumber` | Get field name, given field number in structure array |

| | |
|---|---|
| `mxGetFieldNumber` | Get field number, given field name in structure array |
| `mxGetImagData` | Get pointer to imaginary data of `mxArray` |
| `mxGetInf` | Get value of infinity |
| `mxGetIr` | Get `ir` array of sparse matrix |
| `mxGetJc` | Get `jc` array of sparse matrix |
| `mxGetM` | Get number of rows |
| `mxGetN` | Get number of columns or number of elements |
| `mxGetName` | Get name of specified `mxArray` |
| `mxGetNaN` | Get the value of `NaN` |
| `mxGetNumberOfDimensions` | Get number of dimensions |
| `mxGetNumberOfElements` | Get number of elements in array |
| `mxGetNumberOfFields` | Get number of fields in structure `mxArray` |
| `mxGetNzmax` | Get number of elements in `ir`, `pr`, and `pi` arrays |
| `mxGetPi` | Get `mxArray`'s imaginary data elements |
| `mxGetPr` | Get `mxArray`'s real data elements |
| `mxGetScalar` | Get real component of `mxArray's` first data element |
| `mxGetString` | Copy string `mxArray's` data into C-style string |
| `mxIsCell` | True if cell `mxArray` |
| `mxIsChar` | True if string `mxArray` |
| `mxIsClass` | True if `mxArray` is member of specified class |

| | |
|---|---|
| `mxIsComplex` | True if data is complex |
| `mxIsDouble` | True if `mxArray` represents its data as double-precision, floating-point numbers |
| `mxIsEmpty` | True if `mxArray` is empty |
| `mxIsFinite` | True if value is finite |
| `mxIsFromGlobalWS` | True if `mxArray` was copied from MATLAB's global workspace |
| `mxIsFull (Obsolete)` | Use `mxIsSparse` |
| `mxIsInf` | True if value is infinite |
| `mxIsInt8` | True if `mxArray` represents its data as signed 8-bit integers |
| `mxIsInt16` | True if `mxArray` represents its data as signed 16-bit integers |
| `mxIsInt32` | True if `mxArray` represents its data as signed 32-bit integers |
| `mxIsLogical` | True if `mxArray` is Boolean |
| `mxIsNaN` | True if value is `NaN` |
| `mxIsNumeric` | True if `mxArray` is numeric |
| `mxIsSingle` | True if `mxArray` represents its data as single-precision, floating-point numbers |
| `mxIsSparse` | True if sparse `mxArray` |
| `mxIsString (Obsolete)` | Use `mxIsChar` |
| `mxIsStruct` | True if structure `mxArray` |
| `mxIsUint8` | True if `mxArray` represents its data as unsigned 8-bit integers |
| `mxIsUint16` | True if `mxArray` represents its data as unsigned 16-bit integers |

| | |
|---|---|
| `mxIsUint32` | True if `mxArray` represents its data as unsigned 32-bit integers |
| `mxMalloc` | Allocate dynamic memory using MATLAB's memory manager |
| `mxRealloc` | Reallocate memory |
| `mxRemoveField` | Remove field from structure array |
| `mxSetAllocFcns` | Register memory allocation/deallocation functions in stand-alone engine or MAT application |
| `mxSetCell` | Set value of one cell |
| `mxSetClassName` | Convert MATLAB structure array to MATLAB object array |
| `mxSetData` | Set pointer to data |
| `mxSetDimensions` | Modify number/size of dimensions |
| `mxSetField` | Set field value of structure array, given field name/index |
| `mxSetFieldByNumber` | Set field value in structure array, given field number/index |
| `mxSetImagData` | Set imaginary data pointer for `mxArray` |
| `mxSetIr` | Set `ir` array of sparse `mxArray` |
| `mxSetJc` | Set `jc` array of sparse `mxArray` |
| `mxSetLogical` | Set logical flag |
| `mxSetM` | Set number of rows |
| `mxSetN` | Set number of columns |
| `mxSetName` | Set name of `mxArray` |
| `mxSetNzmax` | Set storage space for nonzero elements |

| | |
|---|---|
| `mxSetPi` | Set new imaginary data for `mxArray` |
| `mxSetPr` | Set new real data for `mxArray` |

# mxAddField

**Purpose**

Add a field to a 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`, `mxSetFieldByNumber`

**Purpose** Convert arrays to strings

**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, mxCreateCharMatrixFromStrings, mxCreateString, mxGetString

# mxAssert

**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 `true`, `mxAssert` does nothing. If `expr` is `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.

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 for debugging purposes; doesn't print assertion's 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 `true`, `mxAssertS` does nothing. If `expr` is `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.

# mxCalcSingleSubscript

**Purpose**

Return the offset (index) from the first element to the 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.

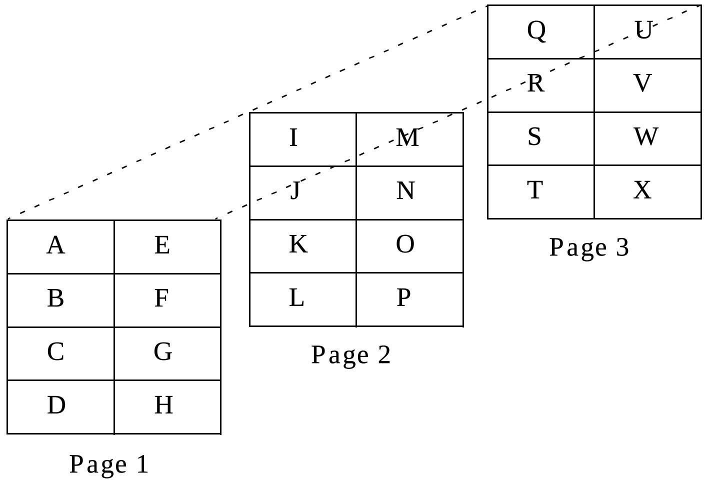
| | |
|---|---|
| A | E |
| B | F |
| C | G |
| D | H |

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 |

Thus, the first column has indices 0 through 3 and the second column has indices 4 through 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 |

Thus, the indices of page 1 are lower than the indices of page 2. Within each page, the indices of the first column are lower than the indices of the second column. Within each column, the indices of the first row are lower than the indices of the second row.

`mxCalcSingleSubscript` provides an efficient way to get an individual offset. However, most applications do not need to get just a single offset. Rather, most applications have to traverse each element of data in an array. In such cases, avoid using `mxCalcSingleSubscript`. To traverse all elements of the array, it is far more efficient to find the array's starting address and then use pointer auto-incrementing to access successive elements. For example, to find the starting address of a numerical array, call `mxGetPr` or `mxGetPi`.

**Example**

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

# mxCalloc

**Purpose** Allocate dynamic memory using MATLAB's 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` defaults to calling the ANSI C `calloc` function. If this default behavior is unacceptable, you can write your own memory allocation routine, and then register this routine with mxSetAllocFcns. Then, whenever mxCalloc is called, mxCalloc calls your memory allocation routine instead of calloc.

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, mxsetallocfcns.c, and mxsetdimensions.c in the mx subdirectory of the examples directory.

## See Also

mxFree, mxDestroyArray, mexMakeArrayPersistent, mexMakeMemoryPersistent, mxMalloc, mxSetAllocFcns

# mxChar

**Purpose**

Data type that string `mxArrays` use to store their data elements

**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`

**Purpose** Enumerated data type that identifies an mxArray's class (category)

**C Syntax**

```
typedef enum {
        mxCELL_CLASS = 1,
        mxSTRUCT_CLASS,
        mxOBJECT_CLASS,
        mxCHAR_CLASS,
        mxSPARSE_CLASS,
        mxDOUBLE_CLASS,
        mxSINGLE_CLASS,
        mxINT8_CLASS,
        mxUINT8_CLASS,
        mxINT16_CLASS,
        mxUINT16_CLASS,
        mxINT32_CLASS,
        mxUINT32_CLASS,
        mxINT64_CLASS, /* place holder - future enhancements */
        mxUINT64_CLASS, /* place holder - future enhancements */
        mxUNKNOWN_CLASS = -1
} mxClassID;
```

**Constants**

mxCELL_CLASS
Identifies a cell mxArray.

mxSTRUCT_CLASS
Identifies a structure mxArray.

mxOBJECT_CLASS
Identifies a user-defined (nonstandard) mxArray.

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

mxSPARSE_CLASS
Identifies a sparse mxArray; that is, an mxArray that only stores its nonzero elements.

`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`
Reserved for possible future use.

`mxUINT64_CLASS`
Reserved for possible future use.

`mxUNKNOWN_CLASS = -1`
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.

## Description

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

## Example

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

## See Also

`mxCreateNumericArray`

**Purpose**

Clear the logical flag

**C Syntax**

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

**Arguments**

`array_ptr`
Pointer to an `mxArray` having a numeric class.

**Description**

Use `mxClearLogical` to turn off the `mxArray`'s logical flag. This flag tells MATLAB that the `mxArray's` data is to be treated as numeric data rather than as Boolean data. If the logical flag is on, then MATLAB treats a 0 value as meaning false and a nonzero value as meaning true.

Call `mxSetLogical` to turn on the `mxArray`'s logical flag. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

**Example**

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

**See Also**

`mxIsLogical`, `mxSetLogical`

# mxComplexity

**Purpose**

Flag that specifies whether an `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`.

**Example**

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

**See Also**

`mxCreateNumericArray`, `mxCreateDoubleMatrix`, `mxCreateSparse`

**Purpose**

Create an unpopulated N-dimensional 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. Causes of failure include:

- Insufficient free heap space.
- Specifying a value for ndim that is greater than the number of values in the dims array.

**Description**

Use mxCellArray 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.

**Example**

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

**See Also**

mxCreateCellMatrix, mxGetCell, mxSetCell, mxIsCell

# mxCreateCellMatrix

**Purpose**

Create an unpopulated two-dimensional 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 empty; 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.

**Example**

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

**See Also**

`mxCreateCellArray`

**Purpose** Create an unpopulated N-dimensional 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`.

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

**See Also** `mxCreateCharMatrixFromStrings`, `mxCreateString`

# mxCreateCharMatrixFromStrings

**Purpose**

Create a populated two-dimensional 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`.

**Example**

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

**See Also**

`mxCreateCharArray`, `mxCreateString`, `mxGetString`

**Purpose**

Create an unpopulated two-dimensional, 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`, `mxComplexity`

**V4 Compatible**

This API function is obsolete and is not supported in MATLAB 5 or later. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mxCreateDoubleMatrix
```

instead of

```
mxCreateFull
```

**See Also**

`mxCreateDoubleMatrix`

# mxCreateNumericArray

**Purpose**

Create an unpopulated N-dimensional 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 ndims 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, mxCELL_CLASS, or mxOBJECT_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.

## 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`, `mxCreateDoubleMatrix`, `mxCreateSparse`, `mxCreateString`, `mxComplexity`

# mxCreateNumericMatrix

**Purpose**

Create a numeric matrix and initialize all its 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 `mxSPARSE_CLASS`, `mxDOUBLE_CLASS`, `mxSINGLE_CLASS`, `mxINT8_CLASS`, `mxUINT8_CLASS`, `mxINT16_CLASS`, `mxUINT16_CLASS`, `mxINT32_CLASS`, and `mxUINT32_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 an 2-dimensional `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`

# mxCreateScalarDouble

**Purpose**

Create a scalar, double-precision array initialized to the specified value

**C Syntax**

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

**Arguments**

`value`
The desired value to which you want to initialize the array.

**Returns**

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

**Description**

Call `mxCreateScalarDouble` to create a scalar double `mxArray`. `mxCreateScalarDouble` 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`, `mxCreateDoubleMatrix`

**Purpose**

Create a two-dimensional 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 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 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.

**Example**

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

**See Also**

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

**Purpose** Create a 1-by-n string mxArray initialized to the 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, mxisclass.c, and mxsetallocfcns.c in the mx subdirectory of the examples directory.

**See Also** mxCreateCharMatrixFromStrings, mxCreateCharArray

# mxCreateStructArray

**Purpose**

Create an unpopulated N-dimensional 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 ndims 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.

**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.

**Example**

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

**See Also** `mxDestroyArray`, `mxSetNzmax`

# mxCreateStructMatrix

**Purpose**

Create an unpopulated two-dimensional 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.

**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.

**Example**

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

**See Also**

mxCreateStructArray, mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber, mxIsStruct

**Purpose**

Free dynamic memory allocated by an mxCreate routine

**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 and mxsetallocfcns.c in the mx subdirectory of the examples directory.

**See Also**

mxCalloc, mxFree, mexMakeArrayPersistent, mexMakeMemoryPersistent

# mxDuplicateArray

**Purpose** Make a deep copy of an array

**C Syntax**

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

**Arguments**

`in`
Pointer to the array's copy.

**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**

Free dynamic memory allocated by `mxCalloc`

**C Syntax**

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

**Arguments**

`ptr`
Pointer to the beginning of any memory parcel allocated by `mxCalloc`.

**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.

By default, when `mxFree` appears in stand-alone MATLAB applications, `mxFree` simply calls the ANSI C `free` function. If this default behavior is unacceptable, you can write your own memory deallocation routine and register this routine with `mxSetAllocFcns`. Then, whenever `mxFree` is called, `mxFree` calls your memory allocation routine instead of `free`.

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`, `mxsetallocfcns.c`, and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

## See Also

`mxCalloc`, `mxDestroyArray`, `mxMalloc`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

## V4 Compatible

This API function is obsolete and is not supported in MATLAB 5 or later. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mxDestroyArray
```

instead of

```
mxFreeMatrix
```

## See Also

`mxDestroyArray`

# mxGetCell

**Purpose**

Get a cell's contents

**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.

**Returns**

A pointer to the `ith` 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.

---

**Example**

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

**See Also**

`mxCreateCellArray`, `mxIsCell`, `mxSetCell`

**Purpose**

Get (as an enumerated constant) an mxArray's class

**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:

`mxCELL_CLASS`
Identifies a cell `mxArray`.

`mxSTRUCT_CLASS`
Identifies a structure `mxArray`.

`mxOBJECT_CLASS`
Identifies a user-defined (nonstandard) `mxArray`.

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

`mxSPARSE_CLASS`
Identifies a sparse `mxArray`; that is, an `mxArray` that only stores its nonzero elements.

`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`
Reserved for possible future use.

`mxUINT64_CLASS`
Reserved for possible future use.

`mxUNKNOWN_CLASS = -1`
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.

## 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 sparse `mxArray`, then `mxGetClassID` returns `mxSPARSE_CLASS`.

`mxGetClassID` is similar to `mxGetClassName`, except that the former returns the class as an enumerated value 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`

**Purpose** Get (as a string) an mxArray's class

**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 sparse mxArray, then mxGetClassName returns sparse.

mxGetClassID is similar to mxGetClassName, except that the former returns the class as an enumerated value 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

# mxGetData

**Purpose**

Get 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 *`. Use `mxGetData` on numeric arrays with contents other than `double`.

**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**

`mxGetPr`

**Purpose** Get a pointer to the 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`

# mxGetElementSize

**Purpose**

Get the 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`, `mxGetN`

**Purpose** Get 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 MATLAB's `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. MATLAB's `PINV` and `RANK` functions use `eps` as a default tolerance.

**Example** See `mxgeteps.c` in the `mx` subdirectory of the `examples` directory.

**See Also** `mxGetInf`, `mxGetNaN`

# mxGetField

## Purpose

Get a field value, given a field name and an index in a 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, and `NULL` otherwise. One possibility is that there is no value assigned to the specified field. Another possibility is that there is a value, but the call failed. 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
```

`mxGetFieldByIndex` is similar to `mxGetField`. Both functions return the same value. The only difference is in the way you specify the field. `mxGetFieldByIndex` 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, mxGetFieldNameByNumber, mxGetFieldNumber, mxGetNumberOfFields, mxIsStruct, mxSetField, mxSetFieldByNumber

# mxGetFieldByNumber

**Purpose**

Get a field value, given a field number and an index in a 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 is a structure `mxArray`.
- Specifying an index < 0 or >= the number of elements in the array.
- Specifying a nonexistent field number. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to determine existing field names.

**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`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxSetField`, `mxSetFieldByNumber`

# mxGetFieldNameByNumber

**Purpose**

Get a field name, given a field number in a 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 `name`; `field_number` 1 represents field name `billing`; `field_number` 2 represents field name `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`, `mxIsStruct`, `mxSetField`

# mxGetFieldNumber

**Purpose**

Get a field number, given a field name in a 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` `"name"` has a field number of 0; the `field_name` `"billing"` has a `field_number` of 1; and the `field_name` `"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.

**Example**

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

**See Also**

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

# mxGetImagData

**Purpose** Get pointer to imaginary data of an 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 *. Use mxGetImagData on numeric arrays with contents other than double.

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

**See Also** mxGetPi

**Purpose** Get the 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.

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

**See Also** `mxGetEps`, `mxGetNaN`

# mxGetIr

## Purpose

Get the ir array of a 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, mxGetNzmax, mxSetIr, mxSetJc, mxSetNzmax

**Purpose**

Get the jc array of a 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, mxSetIr, mxSetJc

# mxGetM

**Purpose**

Get the number of rows

**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`, `mxSetM`, `mxSetN`

**Purpose**

Get the total number of columns in a two-dimensional mxArray or the total number of elements in dimensions 2 through N for an m-by-n array.

**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, mxGetNumberOfDimensions, mxSetM, mxSetN

# mxGetName

**Purpose**

Get the name of the specified `mxArray`

**C Syntax**

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

**Arguments**

`array_ptr`
Pointer to an `mxArray`.

**Returns**

A pointer to the start of the name field. If the `mxArray` has no name, the first element in the name field is \0.

**Description**

Use `mxGetName` to determine the name of the `mxArray` that `array_ptr` points to.

The returned name is a `NULL`-terminated character string. MATLAB variable names are stored in fixed-length character arrays of length `mxMAXNAM+1`, where `mxMAXNAM` is defined in the file `mxArray.h`. Thus variable names can by any length up to `mxMAXNAM`. The actual length is determined by the `NULL` terminator.

`mxGetName` passes back a pointer to an existing section of memory; therefore, your application should not allocate space to hold the returned name string. Do not attempt to deallocate or free the returned string.

**Examples**

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

**See Also**

`mxSetName`

**Purpose**

Get the 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.

**Example**

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

**See Also**

`mxGetEps`, `mxGetInf`

# mxGetNumberOfDimensions

**Purpose**

Get the number of dimensions

**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`, `mxSetN`

**Purpose** Get number of elements in an array

**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 “pieces” an array has. Use `mxGetClassID` to find out what the pieces are. These two functions provide the highest-level information about an array.

**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`, `mxGetM`, `mxGetN`, `mxGetClassID`, `mxGetClassName`

# mxGetNumberOfFields

**Purpose**

Get the number of fields in a 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`, `mxIsStruct`, `mxSetField`

**Purpose** Get the number of elements in the ir, pr, and (if it exists) 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

# mxGetPi

**Purpose** Get an mxArray's imaginary data elements

**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, mxSetPi, mxSetPr

## Purpose

Get an mxArray's real data elements

## 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, it is fairly easy to 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, mxSetPi, mxSetPr

# mxGetScalar

**Purpose**

Get the real component of an mxArray's first data element

**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`, `mxGetN`

# mxGetString

## Purpose

Copy a string mxArray's data into a 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 either point to 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.

---

**Note** Users of multibyte character sets should be aware that MATLAB packs multibyte characters into an mxChar (16-bit unsigned integer). When allocating space for the return string, to avoid possible truncation you should set

```
buflen = (mxGetM(prhs[0] * mxGetN(prhs[0]) * sizeof(mxChar)) + 1
```

---

## 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.

### 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` and `mxsetallocfcns.c` in the `mx` subdirectory of the `examples` directory.

### See Also

`mxCreateCharArray`, `mxCreateCharMatrixFromStrings`, `mxCreateString`

# mxIsCell

**Purpose**

True if a cell mxArray

**C Syntax**

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

**Arguments**

array_ptr
Pointer to an array.

**Returns**

true if array_ptr points to an array having the class mxCELL_CLASS, and false otherwise.

**Description**

Use mxIsCell to determine if the specified array is a cell array.

Do not confuse a cell array with a cell element. Remember that a cell array contains various cell elements, and that most cell elements are not cell arrays.

Calling mxIsCell is equivalent to calling

```
mxGetClassID(array_ptr) == mxCELL_CLASS
```

**See Also**

mxIsClass

**Purpose** True if a string mxArray

**C Syntax**

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

**Arguments**

array_ptr
Pointer to an mxArray.

**Returns**

true if array_ptr points to an array having the class mxCHAR_CLASS, and 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, mxGetClassID

# mxIsClass

**Purpose**

True if mxArray is a member of the 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 enumerated constant). You can specify any one of the following predefined constants:

| Value of Name | Corresponding Class |
|---|---|
| double | mxDOUBLE_CLASS |
| sparse | mxSPARSE_CLASS |
| char | mxCHAR_CLASS |
| cell | mxCELL_CLASS |
| struct | mxSTRUCT_CLASS |
| single | mxSINGLE_CLASS |
| int8 | mxINT8_CLASS |
| uint8 | mxUINT8_CLASS |
| int16 | mxINT16_CLASS |
| uint16 | mxUINT16_CLASS |
| int32 | mxINT32_CLASS |
| uint32 | mxUINT32_CLASS |
| <*class_name*> | mxOBJECT_CLASS |
| unknown | mxUNKNOWN_CLASS |

In the table, <*class_name*> represents the name of a sepcific MATLAB or 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

`true` if `array_ptr` points to an array having category `name`, and `false` otherwise.

### Description

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

### Example

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

### See Also

`mxIsEmpty`, `mxGetClassID`, `mxClassID`

# mxIsComplex

**Purpose** True if data is complex

**C Syntax**

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

**Returns** true if array_ptr is a numeric array containing complex data, and 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.

When a MEX-file is called, MATLAB automatically examines all the input (right-hand side) arrays. If any input array is complex, then MATLAB automatically allocates memory to hold imaginary data for all other input arrays. For example, suppose you pass three input variables (apricot, banana, and carambola) to a MEX-file named Jest:

```
apricot = 7;
banana = sqrt(-5:5);
carambola = magic(2);
Jest(apricot, banana, carambola);
```

banana is complex. Therefore, even though array apricot is purely real, MATLAB automatically allocates space (one element) to hold an imaginary value of apricot. MATLAB also automatically allocates space (four-elements) to hold the nonexistent imaginary values of carambola.

In other words, MATLAB forces every input array to be real or every input array to be complex.

**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`

# mxIsDouble

**Purpose** True if mxArray represents its 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**

true if the mxArray stores its data as double-precision, floating-point numbers, and 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, mxGetClassID

**Purpose** True if `mxArray` is empty

**C Syntax**

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

**Arguments**

`array_ptr`
Pointer to an array.

**Returns** `true` if the `mxArray` is empty, and `false` otherwise.

**Description** Use `mxIsEmpty` to determine if an `mxArray` is empty. An `mxArray` is empty if the size of any of its dimensions is 0.

Attempts to access empty `mxArray` cause undesirable behavior. To avoid accessing empty arrays, test them by calling `mxIsEmpty`.

Note that `mxIsEmpty` is not the opposite of `mxIsFull`.

**Example** See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

**See Also** `mxIsClass`

# mxIsFinite

**Purpose**

True if value is finite

**C Syntax**

```
#include "matrix.h"
bool mxIsFinite(double value);
```

**Arguments**

`value`
The double-precision, floating-point number that you are testing.

**Returns**

`true` if `value` is finite, and `false` otherwise.

**Description**

Call `mxIsFinite` to determine whether or not `value` is finite. A number is finite if it is not equal to `Inf` or `NaN`.

**Examples**

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

**See Also**

`mxIsInf`, `mxIsNaN`

**Purpose** True if the mxArray was copied from MATLAB's global workspace

**C Syntax**

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

**Arguments** array_ptr
Pointer to an mxArray.

**Returns** true if the array was copied out of the global workspace, and false otherwise.

**Description** mxIsFromGlobalWS is useful for stand-alone MAT and engine 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

# mxIsFull (Obsolete)

**V4 Compatible**

This API function is obsolete and is not supported in MATLAB 5 or later. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
if(!mxIsSparse(prhs[0]))
```

instead of

```
if(mxIsFull(prhs[0]))
```

**See Also**

`mxIsSparse`

**Purpose** True if value is infinite

**C Syntax**

```
#include "matrix.h"
bool mxIsInf(double value);
```

**Arguments**

`value`
The double-precision, floating-point number that you are testing.

**Returns** `true` if value is infinite, and `false` otherwise.

**Description** Call `mxIsInf` to determine whether or not `value` is equal to infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents 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.

If `value` equals `NaN` (Not-a-Number), then `mxIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

**Example** See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

**See Also** `mxIsFinite`, `mxIsNaN`

# mxIsInt8

**Purpose**

True if mxArray represents its 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**

`true` if the array stores its data as signed 8-bit integers, and `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`, `mxGetClassID`

**Purpose** True if mxArray represents its 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** true if the array stores its data as signed 16-bit integers, and 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, mxGetClassID

# mxIsInt32

**Purpose**

True if mxArray represents its 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**

true if the array stores its data as signed 32-bit integers, and 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, mxGetClassID

**Purpose**

True if mxArray is Boolean

**C Syntax**

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

**Arguments**

array_ptr
Pointer to an mxArray.

**Returns**

true if the mxArray's logical flag is on, and false otherwise. If an mxArray does not hold numerical data (for instance, if array_ptr points to a structure mxArray or a cell mxArray), then mxIsLogical automatically returns False.

**Description**

Use mxIsLogical to determine whether MATLAB treats the data in the mxArray as Boolean (logical) or numerical (not 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.

**Example**

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

**See Also**

mxIsClass, mxSetLogical

# mxIsNaN

**Purpose** True if value 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** `true` if `value` is `NaN` (Not-a-Number), and `false` otherwise.

**Description** Call `mxIsNaN` to determine whether or not `value` is equal to `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 being equivalent to `NaN`. In other words, `NaN` is not a single value, rather it is a family of numbers that MATLAB (and other IEEE-compliant applications) interpret as being equal to Not-a-Number.

**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`, `mxIsInf`

**Purpose**

True if `mxArray` is numeric

**C Syntax**

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

**Arguments**

`array_ptr`
Pointer to an `mxArray`.

**Returns**

`true` if the array's storage type is:

- `mxDOUBLE_CLASS`
- `mxSPARSE_CLASS`
- `mxSINGLE_CLASS`
- `mxINT8_CLASS`
- `mxUINT8_CLASS`
- `mxINT16_CLASS`
- `mxUINT16_CLASS`
- `mxINT32_CLASS`
- `mxUINT32_CLASS`

`false` if the array's storage type is:

- `mxCELL_CLASS`
- `mxCHAR_CLASS`
- `mxOBJECT_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 `false`. Otherwise, `mxIsNumeric` returns `true`.

Call `mxGetClassID` to determine the exact storage type.

**Example**

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

**See Also**

`mxGetClassID`

# mxIsSingle

**Purpose**

True if `mxArray` represents its 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**

`true` if the array stores its data as single-precision, floating-point numbers, and `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`, `mxGetClassID`

**Purpose** True if a sparse mxArray

**C Syntax**

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

**Arguments**

array_ptr
Pointer to an mxArray.

**Returns** true if array_ptr points to a sparse mxArray, and 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, mxGetJc

# mxIsString (Obsolete)

**V4 Compatible**

This API function is obsolete and is not supported in MATLAB 5 or later. If you need to use this function in existing code, use the `-V4` option of the `mex` script.

Use

```
mxIsChar
```

instead of

```
mxIsString
```

**See Also**

`mxChar`, `mxIsChar`

**Purpose**

True if a structure `mxArray`

**C Syntax**

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

**Arguments**

`array_ptr`
Pointer to an `mxArray`.

**Returns**

`true` if `array_ptr` points to a structure `array`, and `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.

**Example**

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

**See Also**

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetNumberOfFields`, `mxGetField`, `mxSetField`

# mxIsUint8

## Purpose

True if mxArray represents its data as unsigned 8-bit integers

## C Syntax

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

## Arguments

array_ptr
Pointer to an mxArray.

## Returns

true if the mxArray stores its data as unsigned 8-bit integers, and false otherwise.

## Description

Use mxIsInt8 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

mxGetClassID, mxIsClass, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint16, mxIsUint32

**Purpose** True if mxArray represents its 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**

true if the mxArray stores its data as unsigned 16-bit integers, and 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**

mxGetClassID, mxIsClass, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint16, mxIsUint32

# mxIsUint32

**Purpose**

True if `mxArray` represents its 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**

`true` if the `mxArray` stores its data as unsigned 32-bit integers, and `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`, `mxGetClassID`, `mxIsUint16`, `mxIsUint8`, `mxIsInt32`, `mxIsInt16`, `mxIsInt8`

**Purpose** Allocate dynamic memory using MATLAB's 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` defaults to calling the ANSI C `malloc` function. If this default behavior is unacceptable, you can write your own memory allocation routine, and then register this routine with `mxSetAllocFcns`. Then, whenever `mxMalloc` is called, `mxMalloc` calls your memory allocation routine instead of `malloc`.

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`, `mxFree`, `mxDestroyArray`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxSetAllocFcns`

**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`, or by a previous call to `mxRealloc`.

`size`
New size of allocated memory, in bytes.

**Description**

`mxRealloc` reallocates the memory routine for the managed list. If `mxRealloc` fails to allocate a block, you must free the block since the ANSI definition of `realloc` states that the block remains allocated. `mxRealloc` returns `NULL` in this case, and in subsequent calls to `mxRealloc` of the form:

```
x = mxRealloc(x, size);
```

---

**Note** Failure to reallocate memory with `mxRealloc` can result in memory leaks.

---

**Example**

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

**See Also**

`mxCalloc`, `mxFree`, `mxMalloc`, `mxSetAllocFcns`

# mxRemoveField

**Purpose**

Remove a field from a structure array

**C Syntax**

```
#include "matrix.h"
extern void mxRemoveField(mxArray array_ptr, int field_num);
```

**Arguments**

`array_ptr`
Pointer to a structure `mxArray`.

`field_num`
The number of the field you want to remove.

**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.

**See Also**

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

**Purpose**

Register your own memory allocation and deallocation functions in a stand-alone engine or MAT application

**C Syntax**

```
#include "matrix.h"
#include <stdlib.h>
void mxSetAllocFcns(calloc_proc callocfcn, free_proc freefcn,
    realloc_proc reallocfcn, malloc_proc mallocfcn);
```

**Arguments**

`callocfcn`
The name of the function that `mxCalloc` uses to perform memory allocation operations. The function you specify is ordinarily a wrapper around the ANSI C `calloc` function. The `callocfcn` you write must have the prototype:

```
void * callocfcn(size_t nmemb, size_t size);
```

| | |
|---|---|
| `nmemb` | The number of contiguous elements that you want the matrix library to allocate on your behalf. |
| `size` | The size of each element. To get the size, you typically use the `sizeof` operator or the `mxGetElementSize` routine. |

The `callocfcn` you specify must create memory in which all allocated memory has been initialized to zero.

`freefcn`
The name of the function that `mxFree` uses to perform memory deallocation (freeing) operations. The `freefcn` you write must have the prototype:

```
void freefcn(void *ptr);
```

| | |
|---|---|
| `ptr` | Pointer to beginning of the memory parcel to deallocate. |

The `freefcn` you specify must contain code to determine if `ptr` is `NULL`. If `ptr` is `NULL`, then your `freefcn` must not attempt to deallocate it.

`reallocfcn`
The name of the function that `mxRealloc` uses to perform memory reallocation operations. The `reallocfcn` you write must have the prototype:

```
void * reallocfcn(void *ptr, size_t size);
```

| | |
|---|---|
| `ptr` | Pointer to beginning of the memory parcel to reallocate. |
| `size` | The size of each element. To get the size, you typically use the `sizeof` operator or the `mxGetElementSize` routine. |

`mallocfcn`
The name of the function that API functions call in place of `malloc` to perform memory reallocation operations. The `mallocfcn` you write must have the prototype:

```
void * mallocfcn(size_t n);
```

| | |
|---|---|
| `n` | The number of bytes to allocate. |

The `mallocfcn` you specify doesn't need to initialize the memory it allocates.

## Description

Call `mxSetAllocFcns` to establish your own memory allocation and deallocation routines in a stand-alone (nonMEX) application.

It is illegal to call `mxSetAllocFcns` from a MEX-file; doing so causes a compiler error.

In a stand-alone application, if you do not call `mxSetAllocFcns`, then

- `mxCalloc` simply calls the ANSI C `calloc` routine.
- `mxFree` calls a free function, which calls the ANSI C `free` routine if a `NULL` pointer is not passed.
- `mxRealloc` simply calls the ANSI C `realloc` routine.

Writing your own `callocfcn`, `mallocfcn`, `freefcn`, and `reallocfcn` allows you to customize memory allocation and deallocation.

## Example

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

## See Also

`mxCalloc`, `mxFree`, `mxMalloc`, `mxRealloc`

**Purpose** Set the value of one cell

**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` 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`. Use `mxSetCell` to assign new values to unpopulated cells or to overwrite the value of an existing cell.

If the specified cell is already occupied, then `mxSetCell` assigns the new `value`. However, the old cell value remains in memory until you call `mxDestroyArray`.

---

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

**See Also** `mxCreateCellArray`, `mxCreateCellMatrix`, `mxGetCell`, `mxIsCell`

# mxSetClassName

**Purpose**

Convert a MATLAB structure array to a MATLAB object array by specifying a class name to associate with the object

**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`, `mxGetClassID`

## 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 it returns a `void *`. Use this on numeric arrays with contents other than `double`.

## See Also

`mxSetPr`

# mxSetDimensions

**Purpose**

Modify the number of dimensions and/or the size of each dimension

**C Syntax**

```
#include "matrix.h"
int mxSetDimensions(mxArray *array_ptr, const int *dims, int ndims);
```

**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.

`ndims`
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.

**Example**

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

**See Also**

`mxGetNumberOfDimensions`, `mxSetM`, `mxSetN`

**Purpose**

Set a field value of a structure array, given a field name and an 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;
```

If there is already a value at the given position, the `value` pointer you specified overwrites the old value pointer. However, `mxSetField` does not free the dynamic memory that the old value pointer pointed to. Consequently, you should free this old `mxArray` immediately before or after calling `mxSetField`.

---

**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);
```

**Example**

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

**See Also**

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

# mxSetFieldByNumber

**Purpose**

Set a field value in a structure array, given a field number and an 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.

---

**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.

---

**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.

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);
```

## 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`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`

**Purpose**

Set imaginary data pointer for an 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 leaks and other memory errors may result.

**Description**

mxSetImagData is similar to mxSetPi, except it returns a void *. Use this on numeric arrays with contents other than double.

**Example**

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

**See Also**

mxSetPi

# mxSetIr

**Purpose** Set the `ir` array of a 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.

## 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`, `mxGetIr`, `mxGetJc`, `mxSetJc`

# mxSetJc

## Purpose

Set the jc array of a 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. The values in the jc array have the meanings:

- jc[j] is the index in ir, pr (and pi if it exists) of the first nonzero entry in the jth column.
- jc[j+1]-1 is the index of the last nonzero entry in the jth column.
- jc[number of columns + 1] is equal to nnz, which is the number of nonzero entries in the entire spare mxArray.

The number of nonzero elements in any column (denoted as column C) is

```
jc[C] - jc[C-1];
```

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 entries, at `ir[0]`,`ir[1]` |
| `(5,1)` | `4` | `1` | `2` | Column 2 contains one entry, at `ir[2]` |
| `(3,2)` | `2` | `1` | `3` | Column 3 contains three entries, at `ir[3]`,`ir[4]`, `ir[5]` |
| `(2,3)` | `1` | `2` | `6` | There are six nonzero elements. |
| `(5,3)` | `4` | `1` | | |
| `(6,3)` | `5` | `1` | | |

As an example of a much sparser `mxArray`, consider an 8,000 element 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 zero entries |
| `(50,3)` | `49` | `1` | `0` | Column 2 contains one entry, at `ir[0]` |
| `(64,5)` | `63` | `1` | `1` | Column 3 contains one entry, at `ir[1]` |
| | | | `2` | Column 4 contains zero entries. |
| | | | `2` | Column 5 contains one entry, at `ir[3]` |
| | | | `3` | Column 6 contains zero entries. |
| | | | `3` | Column 7 contains zero entries. |
| | | | `3` | Column 8 contains zero entries. |
| | | | `3` | There are three nonzero elements. |

## 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`, `mxGetJc`, `mxSetIr`

**Purpose** Set the logical flag

**C Syntax**

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

**Arguments**

`array_ptr`
Pointer to an `mxArray` having a numeric class.

**Description** Use `mxSetLogical` to turn on an `mxArray`'s logical flag. This flag tells MATLAB that the array's data is to be treated as Boolean. If the logical flag is on, then MATLAB treats a 0 value as meaning `false` and a nonzero value as meaning `true`. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

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

**See Also** `mxClearLogical`, `mxIsLogical`

# mxSetM

**Purpose**

Set the number of rows

**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 array, then you do can optionally reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays in order to use heap space more efficiently.

**Examples**

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

**See Also**

`mxGetM`, `mxGetN`, `mxSetN`

**Purpose** Set the number of columns

**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 size of the `pr`, `pi`, `ir`, or `jc` arrays in order to reduce heap space usage. However, reducing the size is not mandatory.

**Example**

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

**See Also**

`mxGetM`, `mxGetN`, `mxSetM`

# mxSetName

**Purpose**

Set the name of an mxArray

**C Syntax**

```
#include "matrix.h"
void mxSetName(mxArray *array_ptr, const char *name);
```

**Arguments**

array_ptr
Pointer to an mxArray.

name
The name you are assigning to the mxArray. The specified name can be up to mxMAXNAM characters, where mxMAXNAM is a constant defined in the matrix.h header file. If you specify a name longer than mxMAXNAM-1 characters, then mxSetName assigns only the first mxMAXNAM-1 characters to the name.

**Description**

Call mxSetName to establish a name for an mxArray or to change an existing name.

mxSetName assigns the characters in name to a fixed-width section of memory. Do not deallocate this memory.

**Example**

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

**See Also**

mxGetName

**Purpose**

Set the 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.

**Example**

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

# mxSetNzmax

**See Also** `mxGetNzmax`

## Purpose

Set new imaginary data for an 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.

## Examples

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

## See Also

mxSetImagData, mxGetPi, mxGetPr, mxSetPr

# mxSetPr

**Purpose**

Set new real data for an 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 may 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.

**Example**

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

**See Also**

mxGetPr, mxGetPi, mxSetPi

# 6

# Fortran Engine Routines

| | |
|---|---|
| `engClose` | Quit MATLAB engine session |
| `engEvalString` | Evaluate expression in `character` array |
| `engGetFull` | Read full `mxArrays` from engine |
| `engGetMatrix` | Read `mxArrays` from MATLAB engine's workspace |
| `engOpen` | Start MATLAB engine session |
| `engOutputBuffer` | Specify buffer for MATLAB output |
| `engPutFull` | Write full `mxArrays` into workspace of engine |
| `engPutMatrix` | Write `mxArrays` into MATLAB engine's workspace |

**Purpose**

Quit a MATLAB engine session

**Fortran Syntax**

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

**Example**

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

**Purpose** Evaluate expression in `character` array

**Fortran Syntax**

```
integer*4 function engEvalString(ep, command)
integer*4 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 MATLAB's *stdin*. Any output resulting from the command that ordinarily appears on the screen is read back from *stdout* into the buffer defined by `engOutputBuffer`.

**Example**

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

**Purpose**

Read full mxArrays from an engine

**Fortran Syntax**

```
integer*4 function engGetFull(ep, name, m, n, pr, pi)
integer*4 ep, m, n, pr, pi
character*(*) name
```

**Arguments**

ep
Engine pointer.

name
Name of mxArray to get or put into engine's workspace.

m
Row dimension.

n
Column dimension.

pr
Pointer to real part.

pi
Pointer to imaginary part.

**Description**

Most MATLAB applications work only with full (nonsparse) mxArrays. This routine provides an easy way to copy a full mxArray from a MATLAB engine process. It offers an alternative to engGetMatrix, which does not require use of the mxArray structure.

engGetFull reads the named mxArray from the engine pointed to by ep and places the row dimensions, column dimensions, real array pointer, and imaginary array pointer into the locations specified by m, n, pr, and pi, respectively.

engGetFull returns 0 if successful, and 1 otherwise.

engGetFull allocates memory for the real and imaginary arrays using mxCalloc; use mxFree to return it when you are done.

If the mxArray is purely real, the imaginary pointer is given 0.

**Note** This routine will become obsolete in a future version. Use `engGetMatrix`, `mxGetPr`, `mxGetPi`, `mxGetM`, and `mxGetN` instead.

**Purpose**

Read mxArrays from a MATLAB engine's workspace

**Fortran Syntax**

```
integer*4 function engGetMatrix(ep, name)
integer*4 ep
character*(*) name
```

**Arguments**

ep
Engine pointer.

name
Name of mxArray to get from engine.

**Description**

This routine allows you to copy an mxArray out of a MATLAB engine's workspace.

engGetMatrix reads the named mxArray from the engine pointed to by ep 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.

On UNIX systems, engGetMatrix issues the command save stdio name to MATLAB, causing MATLAB to write the named mxArray down its *stdout* pipe, which is in turn caught and decoded by engGetMatrix.

**Example**

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.

# engOpen

**Purpose** Start a MATLAB engine session

**Fortran Syntax**

```
integer*4 function engOpen(startcmd)
integer*4 ep
character*(*) startcmd
```

**Arguments**

`ep`
Engine pointer.

`startcmd`
`Character` array to start 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).

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

**Purpose**

Specify buffer for MATLAB output

**Fortran Syntax**

```
integer*4 function engOutputBuffer(ep, p)
integer*4 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.

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`.

**Example**

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.

# engPutFull

**Purpose** Write full mxArrays into the workspace of an engine

**Fortran Syntax**

```
integer*4 function engPutFull(ep, name, m, n, pr, pi)
integer*4 ep, m, n, pr, pi
character*(*) name
```

**Arguments**

ep
Engine pointer.

name
Name of mxArray to put into engine's workspace.

m
Row dimension.

n
Column dimension.

pr
Pointer to real part.

pi
Pointer to imaginary part.

**Description** Most MATLAB applications work only with full (nonsparse) mxArrays. This routine provides an easy way to write a full mxArray into a MATLAB engine process. It offers an alternative to engPutMatrix, which does not require use of the mxArray structure.

engPutFull writes the mxArray with dimensions m-by-n, real data pr, and imaginary data pi into the workspace of engine ep with the specified name.

If the mxArray does not exist in the engine's 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.

---

**Note** This routine will become obsolete in a future version. Use engPutMatrix, mxSetPr, mxSetPi, mxSetM, and mxSetN instead.

---

**Purpose** Write mxArrays into a MATLAB engine's workspace

**Fortran Syntax**

```
integer*4 function engPutMatrix(ep, mp)
integer*4 mp, ep
```

**Arguments**

ep
Engine pointer.

mp
mxArray pointer.

**Description** This routine allows you to write an mxArray into a MATLAB engine's workspace.

engPutMatrix writes mxArray mp 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.

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

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

On UNIX systems, engPutMatrix issues the command load stdio name to MATLAB and sends the data down the *stdin* pipe.

**Example** 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.

7

# Fortran MAT-File Routines

| | |
|---|---|
| `matClose` | Close MAT-file |
| `matDeleteMatrix` | Delete named `mxArray` from MAT-file |
| `matGetDir` | Get directory of `mxArrays` in MAT-file |
| `matGetFull` | Read full `mxArrays` from MAT-file |
| `matGetMatrix` | Read `mxArrays` from MAT-file |
| `matGetNextMatrix` | Get next `mxArray` from MAT-file |
| `matGetString` | Copy `character` `mxArrays` from MAT-file |
| `matOpen` | Open MAT-file |
| `matPutFull` | Write full `mxArrays` into MAT-file |
| `matPutMatrix` | Write `mxArrays` into MAT-file |
| `matPutString` | Write `character` `mxArrays` into MAT-file |

# matClose

**Purpose** Closes a MAT-file

**Fortran Syntax**

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

**Purpose**

Delete named `mxArray` from MAT-file

**Fortran Syntax**

```
subroutine matDeleteMatrix(mfp, name)
integer*4 mfp
character*(*) name
```

**Arguments**

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray` to delete.

**Description**

`matDeleteMatrix` deletes the named `mxArray` from the MAT-file pointed to by `mfp`. The file is rewritten to accomplish this task. `matDeleteMatrix` returns 0 if successful, and nonzero if an error occurs.

**Example**

See `matdemo1.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.

# matGetDir

**Purpose**

Get directory of mxArrays in a MAT-file

**Fortran Syntax**

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

**Example**

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.

# matGetFull

**Purpose** Reads full mxArrays from MAT-files

**Fortran Syntax**

```
integer*4 function matGetFull(mfp, name, m, n, pr, pi)
integer*4 mfp, m, n, pr, pi
character*(*) name
```

**Arguments**

mfp
Pointer to MAT-file information.

name
Name of mxArray to get or put to MAT-file.

m
Row dimension.

n
Column dimension.

pr
Pointer to real part.

pi
Pointer to imaginary part.

**Description**

Most MATLAB applications work only with full (nonsparse) mxArrays. This routine provides an easy way to copy a full mxArray out of a MAT-file. It offers an alternative to matGetMatrix, which does not require use of the mxArray structure.

matGetFull reads the named mxArray from the MAT-file pointed to by mfp and places the row dimensions, column dimensions, real array pointer, and imaginary array pointer into the locations specified by m, n, pr, and pi, respectively.

matGetFull returns 0 if successful, and 1 if the named variable can't be found, the named variable is not a full mxArray, or there is a file read error.

matGetFull allocates memory for the real and imaginary arrays using mxCalloc; use mxFree to return the memory when you are done.

If the mxArray is pure real, the imaginary pointer is 0.

**Note** This routine will become obsolete in a future version. Use `matGetMatrix`, `mxGetPr`, `mxGetPi`, `mxGetM`, and `mxGetN` instead.

**Purpose**

Reads mxArrays from MAT-files

**Fortran Syntax**

```
integer*4 function matGetMatrix(mfp, name)
integer*4 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.

matGetMatrix 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.

**Example**

See matdemo1.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.

# matGetNextMatrix

**Purpose**

Get next mxArray from MAT-file

**Fortran Syntax**

```
integer*4 function matGetNextMatrix(mfp)
integer*4 mfp
```

**Arguments**

`mfp`
Pointer to MAT-file information.

**Description**

This routine allows you to step sequentially through a MAT-file and read all the `mxArrays` in a single pass.

`matGetNextMatrix` reads the next `mxArray` from the MAT-file pointed to by `mfp` and returns a pointer to a newly allocated `mxArray` structure. Use it 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.

`matGetNextMatrix` 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.

**Example**

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.

## Purpose

Copy character mxArrays from MAT-files

## Fortran Syntax

```
integer*4 function matGetString(mfp, name, str, strlen)
integer*4 mfp, strlen
character*(*) name, str
```

## Arguments

`mfp`
Pointer to MAT-file information.

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

`str`
`character` array to read from MAT-file.

`strlen`
Length of the `character` array.

## Description

`matGetString` reads the `character` `mxArray` with the specified `name` into `str` from the MAT-file `mfp`. It returns zero if successful, and a nonzero value if an error occurs.

`matGetString` copies the `character` array from `mxArray` `name` on file `mfp` into the character array `str`.

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. If the `character` `mxArray` contains several rows, they are copied, one column at a time, into one long `character` array.

`matGetString` returns 0 if the copy is successful, and 1 if the copy has failed because the `mxArray` is not a `character` `mxArray`, 2 if the length of the `character` array exceeds `strlen`, and 3 if there is a file read error.

## Example

```
      program main
      integer matOpen, matClose, matPutString
      integer mfp, stat
c
      mfp = matOpen('foo.mat', 'w')
      stat = matPutString(mfp,'A','Hello, world')
      stat = matClose(mfp)
```

```
c
    stop
    end
```

Then you can go to MATLAB and enter:

```
load foo
A
A =
    Hello, world
```

**Purpose**

Opens a MAT-file

**Fortran Syntax**

```
integer*4 function matOpen(filename, mode)
integer*4 mfp
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 MATLAB 4 MAT-file. |

`mfp`
Pointer to MAT-file information.

**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.

# matPutFull

**Purpose** Writes full mxArrays into MAT-files

**Fortran Syntax**

```
integer*4 function matPutFull(mfp, name, m, n, pr, pi)
integer*4 mfp, m, n, pr, pi
character*(*) name
```

**Arguments**

mfp
Pointer to MAT-file information.

name
Name of mxArray to write to MAT-file.

m
Row dimension.

n
Column dimension.

pr
Pointer to real part.

pi
Pointer to imaginary part.

**Description**

Most MATLAB applications work only with full (nonsparse) mxArrays. This routine provides an easy way to write a full mxArray into a MAT-file. It offers an alternative to matPutMatrix, which does not require use of the mxArray structure.

matPutFull writes the mxArray with dimensions m-by-n, real data pr, and imaginary data pi onto the MAT-file mfp with the specified name.

If the mxArray does not exist on 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.

---

**Note** This routine will become obsolete in a future version. Use matPutMatrix, mxSetPr, mxSetPi, mxSetM, and mxSetN instead.

---

**Examples**

Read the mxArray A from one MAT-file and write it out to another.

```
      program main
      integer matOpen,matClose,matPutFull,matGetFull
      integer mf1, mf2, stat
      integer m, n, pr, pi
      mf1 = matOpen('foo.mat','r')
      mf2 = matOpen('foo2.mat','w')
      stat = matGetFull(mf1,'A',m,n,pr,pi)
      stat = matPutFull(mf2,'A',m,n,pr,pi)
      stat = matClose(mf1)
      stat = matClose(mf2)
c
      stop
      end
```

Write a simple real `mxArray` into a MAT-file. Name the `mxArray` `A` and the MAT-file `foo.mat`.

```
      integer matOpen, matClose, matPutFull
      integer mfp, stat
      double precision Areal(6)
      data Areal / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /
      data Aimag / 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 /
c
      mfp = matOpen('foo.mat','w')
      stat = matPutFull(mfp,'A',3,2,Areal,Aimag)
      stat = matClose(mfp)
c
      stop
      end
```

To test, run the second example; then go to MATLAB and enter:

```
load foo
A
A =
   1   4
   2   5
   3   6
```

# matPutMatrix

**Purpose**

Writes mxArrays into MAT-files

**Fortran Syntax**

```
integer*4 function matPutMatrix(mfp, mp)
integer*4 mp, mfp
```

**Arguments**

mfp
Pointer to MAT-file information.

mp
mxArray pointer.

**Description**

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

matPutMatrix 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 than the existing mxArray.

matPutMatrix returns 0 if successful and nonzero if an error occurs.

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

**Example**

See matdemo1.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.

**Purpose**

Write character mxArrays into MAT-files

**Fortran Syntax**

```
integer*4 function matPutString(mfp, name, str)
integer*4 mfp
character*(*) name, str
```

**Arguments**

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray` to write to MAT-file.

`str`
`character` array to write to MAT-file.

**Description**

`matPutString` writes the `mxArray` with the specified `name` and `str` to the MAT-file `mfp`. It returns 0 if successful, and 1 if an error occurs.

If the `mxArray` does not exist on 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.

**Example**

```
      program main
      integer matOpen, matClose, matPutString
      integer mfp, stat
c
      mfp = matOpen('foo.mat', 'w')
      stat = matPutString(mfp,'A','Hello, world')
      stat = matClose(mfp)
c
      stop
      end
```

Then you can go to MATLAB and enter:

```
load foo
A
A =
   Hello, world
```

8

# Fortran MEX-Functions

| | |
|---|---|
| `mexAtExit` | Register function to be called when MATLAB is cleared or terminates |
| `mexCallMATLAB` | Call MATLAB function or user-defined M-file or MEX-file |
| `mexErrMsgTxt` | Issue error message and return to MATLAB |
| `mexEvalString` | Execute MATLAB command in caller's workspace |
| `mexFunction` | Entry point to Fortran MEX-file |
| `mexGetEps` | Get the value of `eps` |
| `mexGetFull` | Get component parts of double-precision `mxArray` into Fortran workspace |
| `mexGetGlobal` | Get pointer to `mxArray` from MATLAB's global workspace |
| `mexGetInf` | Get value of infinity |
| `mexGetMatrix` | Copies `mxArray` from caller's workspace |
| `mexGetMatrixPtr` | Get pointer to `mxArray` in caller's workspace |
| `mexGetNaN` | Get value of `NaN` |
| `mexIsFinite` | Determine whether or not value is finite |
| `mexIsInf` | Determine whether or not value is infinite |
| `mexIsNaN` | Determine whether or not value is `NaN` |
| `mexPrintf` | Print `character` array |
| `mexPutFull` | Create `mxArray` from component parts into Fortran workspace |

| | |
|---|---|
| `mexPutMatrix` | Writes `mxArray` to caller's workspace |
| `mexSetTrapFlag` | Control response of `mexCallMATLAB` to errors |

**Purpose** Register a subroutine to be called when the MEX-file is cleared or when MATLAB terminates

**Fortran Syntax**

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

**Arguments**

`ExitFcn`
The exit function.

**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`

# mexCallMATLAB

**Purpose**

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

**Fortran Syntax**

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

On the Alpha platform, use:

```
integer*8 function mexCallMATLAB(nlhs, plhs, nrhs, prhs, name)
integer*4 nlhs, nrhs
integer*8 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`, `mexSetTrapFlag`

# mexErrMsgTxt

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

**Fortran Syntax**

```
subroutine mexErrMsgTxt(error_msg)
character*(*) error_msg
```

**Arguments**

`error_msg`
`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.

**Purpose**

Execute a MATLAB command in the workspace of the caller

**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`

# mexFunction

**Purpose**

MATLAB entry point to a Fortran MEX-file

**Fortran Syntax**

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer*4 nlhs, nrhs, 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-function 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`.

**Purpose** Get the value of `eps`

**Fortran Syntax**

```
real*8 function mexGetEps()
```

**Arguments** `none`

**Returns** The value of MATLAB's `eps` variable.

**Description** The `eps` variable holds the distance between 1.0 and the next largest floating-point number. It is a measure of floating-point accuracy. MATLAB's `PINV` and `RANK` functions use `eps` as a default tolerance.

**See Also** `mexGetInf`, `mexGetNaN`

# mexGetFull

**Purpose**

Routine to get component parts of a double-precision `mxArray` into a Fortran workspace

**Fortran Syntax**

```
integer*4 function mexGetFull(name, m, n, pr, pi)
integer*4 m, n, pr, pi
character*(*) name
```

**Arguments**

`name`
Name of `mxArray` to get from workspace.

`m`
Row dimension.

`n`
Column dimension.

`pr`
Pointer to real part.

`pi`
Pointer to imaginary part.

**Returns**

0 if successful, and 1 otherwise.

**Description**

`mexGetFull` provides a way to copy data from a double-precision `mxArray` from the caller's workspace. It is an alternative to `mexGetMatrix`, which does not require use of the `mxArray` structure.

`mexGetFull` reads the named `mxArray` from the caller's workspace and places the row dimensions, column dimensions, real array pointer, and imaginary array pointer into the locations specified by `m`, `n`, `pr`, and `pi`, respectively. You can then use `mxCopyPtrToReal8` to copy the data from the pointer into the Fortran workspace.

`mexGetFull` allocates memory for the real and imaginary arrays using `mxCalloc`; use `mxFree` to return it when you are done.

If the `mxArray` is purely real, the imaginary pointer is given `0`.

**See Also**

`mxGetName`, `mxGetPr`, `mxGetPi`

**Purpose**

Get a pointer to an mxArray from MATLAB's global workspace

**Fortran Syntax**

```
integer*4 function mexGetGlobal(name)
character*(*) name
```

**Arguments**

name
Name of mxArray to get from workspace.

**Returns**

Pointer to global mxArray if successful, or 0 if it doesn't exist.

**Description**

mexGetGlobal gets an mxArray from MATLAB's global workspace instead of from the caller's workspace.

**See Also**

mxGetName, mxGetPr, mxGetPi

# mexGetInf

**Purpose**

Get the value of infinity

**Fortran Syntax**

```
real*8 function mexGetInf()
```

**Arguments**

`none`

**Returns**

The value of infinity on your system.

**Description**

Call `mexGetInf` 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 in to 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(100000)` returns infinity because the result is too large to be represented on your machine.

**See Also**

`mexGetEps`, `mexGetNaN`

**Purpose**

Copies an mxArray from the caller's workspace

**Fortran Syntax**

```
integer*4 function mexGetMatrix(name)
character*(*) name
```

**Arguments**

name
Name of mxArray to get from workspace.

**Returns**

A pointer to a newly allocated mxArray if successful. Otherwise, returns 0.

**Description**

mexGetMatrix reads the named mxArray from the caller's workspace, and returns a pointer to a newly allocated mxArray or 0 if the attempt fails.

# mexGetMatrixPtr

**Purpose** Get the pointer to an mxArray in the caller's workspace

**Fortran Syntax**

```
integer*4 function mexGetMatrixPtr(name)
character*(*) name
```

**Arguments**

name
Name of mxArray to get from caller's workspace.

**Returns** A pointer to an mxArray owned by MATLAB.

**Description**

mexGetMatrixPtr returns a pointer to the mxArray with the specified name in the workspace local to the calling function. It allows you to read or modify variables in the MATLAB workspace directly from a MEX-file.

Do not free or reallocate the memory associated with any part of an mxArray obtained with the mexGetMatrixPtr function, including the real part, imaginary part, and sparse structure. mxArrays obtained with this function are managed by MATLAB's own internal mechanisms and MATLAB will crash immediately if you change them.

mexGetMatrixPtr is meant to be used to read values from an mxArray in the workspace or to change those values, provided the mxArray remains the same size, complexity, and sparsity.

To get the pointer of a global variable that is not defined as global by the calling function, first declare it global with a call of the form mexEvalString("global varname").

**Purpose**

Get the value of `NaN` (Not-a-Number)

**Fortran Syntax**

```
real*8 function mexGetNan()
```

**Arguments**

`none`

**Returns**

MATLAB's value of `NaN` (Not-a-Number).

**Description**

Call `mexGetNaN` to return the value of `NaN` for MATLAB. `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`

**See Also**

`mexGetEps`, `mexGetInf`

# mexIsFinite

**Purpose** Determine whether or not a value is finite

**Fortran Syntax**

```
integer*4 function mexIsFinite(value)
real*8 value
```

**Arguments**

`value`
The double-precision, floating-point number you are testing.

**Returns** `true` if value is finite, and `false` otherwise.

**Description** Call `mexIsFinite` to determine whether or not `value` is finite. A number is finite if it is not equal to `Inf` or `NaN`.

**See Also** `mexIsInf`, `mexIsNaN`

**Purpose** Determine whether or not a value is infinite

**Fortran Syntax**

```
integer*4 function mexIsInf(value)
real*8 value
```

**Arguments**

`value`
The double-precision, floating-point number you are testing.

**Returns** `true` if value is infinite, and `false` otherwise.

**Description** Call `mexIsInf` to determine whether or not `value` is equal to infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of `Inf` is built in to 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 `mexIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

**See Also** `mexIsFinite`, `mexIsNaN`

# mexIsNaN

**Purpose**

Determine whether or not a value is `NaN` (Not-a-Number)

**Fortran Syntax**

```
integer*4 function mexIsNaN(value)
real*8 value
```

**Arguments**

`value`
The double-precision, floating-point number you are testing.

**Returns**

`true` if `value` is `NaN` (Not-a-Number), and `false` otherwise.

**Description**

Call `mexIsNaN` to determine whether or not value is equal to `NaN`, 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`

**See Also**

`mexIsFinite`, `mexIsInf`, `mexGetInf`

**Purpose** Print a character array

**Fortran Syntax**

```
subroutine 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.

---

**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`

# mexPutFull

**Purpose**

Routine to create an mxArray from its component parts into a Fortran workspace

**Fortran Syntax**

```
integer*4 function mexPutFull(name, m, n, pr, pi)
integer*4 m, n, pr, pi
character*(*) name
```

**Arguments**

name
Name of mxArray to put into workspace.

m
Row dimension.

n
Column dimension.

pr
Pointer to real part.

pi
Pointer to imaginary part.

**Returns**

0 if successful, and 1 otherwise.

**Description**

Most MATLAB applications work only with full (nonsparse) mxArrays. mexPutFull provides an easy way to write a full mxArray into a MEX-file's caller's workspace. It is an alternative to mexPutMatrix, which requires use of the mxArray structure.

mexPutFull writes the mxArray with dimensions m-by-n, real data pr, and imaginary data pi into the calling workspace with the specified name. If an mxArray with the same name already exists in the workspace, the existing mxArray is replaced with the new one.

**See Also**

mxSetName

**Purpose** Writes an mxArray to the caller's workspace

**Fortran Syntax**

```
integer*4 function mexPutMatrix(mp)
integer*4 mp
```

**Arguments**

mp
Pointer to mxArray.

**Returns** 0 if successful, and 1 if an error occurs.

**Description** mexPutMatrix writes mxArray mp to the caller's workspace. 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 one.

# mexSetTrapFlag

**Purpose** Control response of `mexCallMATLAB` to errors

**Fortran Syntax**

```
subroutine mexSetTrapFlag(trap_flag)
integer*4 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 MATLAB's 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.

**See Also** `mexAtExit`, `mexErrMsgTxt`

9

# Fortran MX-Functions

| | |
|---|---|
| `mxCalloc` | Allocate dynamic memory using MATLAB's memory manager |
| `mxCopyCharacterToPtr` | Copy `character` values from Fortran array to pointer array |
| `mxCopyComplex16ToPtr` | Copy `COMPLEX*16` values from Fortran array to pointer array |
| `mxCopyInteger4ToPtr` | Copy `INTEGER*4` values from Fortran array to pointer array |
| `mxCopyPtrToCharacter` | Copy `character` values from pointer array to Fortran array |
| `mxCopyPtrToComplex16` | Copy `COMPLEX*16` values from pointer array to Fortran array |
| `mxCopyPtrToInteger4` | Copy `INTEGER*4` values from pointer array to Fortran array |
| `mxCopyPtrToPtrArray` | Copy pointer values from pointer array to Fortran array |
| `mxCopyPtrToReal8` | Copy `REAL*8` values from pointer array to Fortran array |
| `mxCopyReal8ToPtr` | Copy `REAL*8` values from Fortran array to pointer array |
| `mxCreateFull` | Create unpopulated two-dimensional `mxArray` |
| `mxCreateSparse` | Create two-dimensional unpopulated sparse `mxArray` |
| `mxCreateString` | Create 1-by-n `character` array initialized to specified string |
| `mxFree` | Free dynamic memory allocated by `mxCalloc` |
| `mxFreeMatrix` | Free dynamic memory allocated by `mxCreateFull` and `mxCreateSparse` |
| `mxGetIr` | Get `ir` array |

| | |
|---|---|
| mxGetJc | Get jc array |
| mxGetM | Get number of rows |
| mxGetN | Get total number of columns |
| mxGetName | Get name of specified mxArray |
| mxGetNzmax | Get number of elements in ir, pr, and pi arrays |
| mxGetPi | Get mxArray's imaginary data elements |
| mxGetPr | Get mxArray's real data elements |
| mxGetScalar | Get real component of mxArray's first data element |
| mxGetString | Create character array from mxArray |
| mxIsComplex | Inquire if mxArray is complex |
| mxIsDouble | Inquire if mxArray is of type double |
| mxIsFull | Inquire if mxArray is full |
| mxIsNumeric | Inquire if mxArray contains numeric data |
| mxIsSparse | Inquire if mxArray is sparse |
| mxIsString | Inquire if mxArray contains character array |
| mxSetIr | Set ir array of sparse mxArray |
| mxSetJc | Set jc array of sparse mxArray |
| mxSetM | Set number of rows |
| mxSetN | Set number of columns |
| mxSetName | Set name of mxArray |

# mxCalloc

**Purpose** Allocate dynamic memory using MATLAB's memory manager

**Fortran Syntax**

```
integer*4 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, MATLAB's memory manager is not used.

**See Also** `mxFree`

**Purpose**

Copy `character` values from a Fortran array to a pointer array

**Fortran Syntax**

```
subroutine mxCopyCharacterToPtr(y, px, n)
character*(*) y
integer*4 px, 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's pointer arrays and ordinary Fortran `character` arrays.

**See Also**

`mxCopyPtrToCharacter`

# mxCopyComplex16ToPtr

**Purpose**

Copy COMPLEX*16 values from a Fortran array to a pointer array

**Fortran Syntax**

```
subroutine mxCopyComplex16ToPtr(y, pr, pi, n)
complex*16 y(n)
integer*4 pr, pi, n
```

**Arguments**

y
COMPLEX*16 Fortran array.

pr
Pointer to pr array.

pi
Pointer to pi 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

# mxCopyInteger4ToPtr

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

**Fortran Syntax**

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

**Arguments**

y
INTEGER*4 Fortran array.

n
Number of elements to copy.

px
Pointer to ir or jc array.

**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

# mxCopyPtrToCharacter

**Purpose**

Copy `character` values from a pointer array to a Fortran array

**Fortran Syntax**

```
subroutine mxCopyPtrToCharacter(px, y, n)
character*(*) y
integer*4 px, 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's pointer arrays into ordinary Fortran `character` arrays.

**See Also**

`mxCopyCharacterToPtr`

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

**Fortran Syntax**

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

**Arguments**

pr
Pointer to pr array.

pi
Pointer to pi 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

# mxCopyPtrToInteger4

**Purpose**

Copy INTEGER*4 values from a pointer array to a Fortran array

**Fortran Syntax**

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

# mxCopyPtrToPtrArray

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

**Fortran Syntax**

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

**Example**

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**

`mxCopyInteger4ToPtr`

# mxCopyPtrToReal8

**Purpose** Copy `REAL*8` values from a pointer array to a Fortran array

**Fortran Syntax**

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

**Arguments**

`px`
Pointer to `pr` or `pi` 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.

**See Also** `mxCopyReal8ToPtr`

**Purpose**

Copy REAL*8 values from a Fortran array to a pointer array

**Fortran Syntax**

```
subroutine mxCopyReal8ToPtr(y, px, n)
real*8 y(n)
integer*4 px, n
```

**Arguments**

y
REAL*8 Fortran array.

px
Pointer to pr or pi 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.

**See Also**

mxCopyPtrToReal8

# mxCreateFull

## Purpose

Create an unpopulated two-dimensional mxArray

## Fortran Syntax

```
integer*4 function mxCreateFull(m, n, ComplexFlag)
integer*4 m, n, ComplexFlag
```

## Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

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

## Returns

An unpopulated, m-by-n `mxArray` if successful, and 0 otherwise.

## Description

Use `mxCreateFull` to create an unpopulated `mxArray` of size m-by-n. `mxCreateFull` initializes each element in the `pr` array to 0. If you set `ComplexFlag` to 1, `mxCreateFull` also initializes each element in the `pi` array to 0.

If you specify `REAL = 0`, `mxCreateFull` allocates enough memory to hold m-by-n real elements. If you specify `COMPLEX = 1`, `mxCreateFull` allocates enough memory to hold m-by-n real elements and m-by-n imaginary elements.

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

## See Also

`mxCreateSparse`, `mxFreeMatrix`

**Purpose**

Create a two-dimensional unpopulated sparse mxArray

**Fortran Syntax**

```
integer*4 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 mxArray if successful, and 0 otherwise.

**Description**

Call mxCreateSparse to create an unpopulated sparse 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 mxFreeMatrix to reclaim all its heap space.

**See Also**

mxFreeMatrix, mxSetNzmax, mxSetPr, mxSetIr, mxSetJc

# mxCreateString

**Purpose** Create a 1-by-n `character` array initialized to the specified string

**Fortran Syntax**

```
integer*4 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 `mxFreeMatrix`.

**Purpose** Free dynamic memory allocated by `mxCalloc`

**Fortran Syntax**

```
subroutine mxFree(ptr)
integer*4 ptr
```

**Arguments**

`ptr`
Pointer to the beginning of any memory parcel allocated by `mxCalloc`.

**Description**

`mxFree` deallocates heap space. `mxFree` frees memory using MATLAB's own 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`, `mxFreeMatrix`

# mxFreeMatrix

**Purpose**

Free dynamic memory allocated by `mxCreateFull` and `mxCreateSparse`

**Fortran Syntax**

```
subroutine mxFreeMatrix(pm)
integer*4 pm
```

**Arguments**

`pm`
Pointer to the beginning of the `mxArray`.

**Description**

`mxFreeMatrix` returns an `mxArray` to the heap for reuse, freeing any arrays (`pr`, `pi`, `ir`, or `jc`) allocated within the `mxArray`.

**See Also**

`mxCalloc`, `mxFree`

**Purpose**

Get the `ir` array

**Fortran Syntax**

```
integer*4 function mxGetIr(pm)
integer*4 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` and `mxSetJc`.

**See Also**

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

# mxGetJc

**Purpose**

Get the jc array

**Fortran Syntax**

```
integer*4 function mxGetJc(pm)
integer*4 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.

**See Also**

mxGetIr, mxSetIr, mxSetJc

**Purpose**

Get the number of rows

**Fortran Syntax**

```
integer*4 function mxGetM(pm)
integer*4 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.

**See Also**

`mxGetN`, `mxSetM`, `mxSetN`

# mxGetN

**Purpose** Get the total number of columns

**Fortran Syntax**

```
integer*4 function mxGetN(pm)
integer*4 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.

**See Also** mxGetM, mxSetM, mxSetN

**Purpose** Get the name of the specified `mxArray`

**Fortran Syntax**

```
character*32 function mxGetName(pm)
integer*4 pm
```

**Arguments**

`pm`
Pointer to an `mxArray`.

**Returns** A pointer to the start of the name field. If the `mxArray` has no name, `mxGetName` returns 0.

**Description** Use `mxGetName` to determine the name of the `mxArray` that `pm` points to. The returned `mxArray` name is a `character` array with maximum length 31.

**See Also** `mxSetName`

# mxGetNzmax

**Purpose** Get the number of elements in the `ir`, `pr`, and (if it exists) `pi` arrays

**Fortran Syntax**

```
integer*4 function mxGetNzmax(pm)
integer*4 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`

**Purpose**

Get an mxArray's imaginary data elements

**Fortran Syntax**

```
integer*4 function mxGetPi(pm)
integer*4 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**

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 mxArrays to a MATLAB function are allocated if any of the input mxArrays is complex.

If you use mxGetPr or mxGetPi, note that mxFreeMatrix frees pr and pi using mxFree, so pr and pi should only be set to memory allocated with mxCalloc.

**See Also**

mxGetPr, mxSetPi, mxSetPr

# mxGetPr

**Purpose**

Get an `mxArray`'s real data elements

**Fortran Syntax**

```
integer*4 function mxGetPr(pm)
integer*4 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**

Call `mxGetPr` to determine the starting address of the real data in the `mxArray` that `pm` points to. Once you have the starting address, it is fairly easy to access any other element in the `mxArray`.

If you use `mxGetPr` or `mxGetPi`, note that `mxFreeMatrix` frees `pr` and `pi` using `mxFree`, so `pr` and `pi` should only be set to memory allocated with `mxCalloc`.

**See Also**

`mxGetPi`, `mxSetPi`, `mxSetPr`

**Purpose** Get the real component of an mxArray's first data element

**Fortran Syntax**

```
real*8 function mxGetScalar(pm)
integer*4 pm
```

**Arguments**

pm
Pointer to an mxArray.

**Returns** The value of the first real (nonimaginary) element of the mxArray. If the mxArray is larger than 1-by-1, mxGetScalar returns the value of the (1,1) element.

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, mxGetN

# mxGetString

**Purpose**

Create a character array from an mxArray

**Fortran Syntax**

```
integer*4 function mxGetString(pm, str, strlen)
integer*4 pm, strlen
character*(*) str
```

**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

**Purpose**

Inquire if an mxArray is complex

**Fortran Syntax**

```
integer*4 function mxIsComplex(pm)
integer*4 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.

When a MEX-file is called, MATLAB automatically examines all the input (right-hand side) arrays. If any input array is complex, then MATLAB automatically allocates memory to hold imaginary data for all other input arrays. For example, suppose you pass three input variables (apricot, banana, and carambola) to a MEX-file named Jest:

```
apricot = 7;
banana = sqrt(-5:5);
carambola = magic(2);
Jest(apricot, banana, carambola);
```

banana is complex. Therefore, even though array apricot is purely real, MATLAB automatically allocates space (one element) to hold an imaginary value of apricot. MATLAB also automatically allocates space (four elements) to hold the nonexistent imaginary values of carambola.

In other words, MATLAB forces every input array to be real or every input array to be complex.

**See Also**

mxIsNumeric

# mxIsDouble

**Purpose** Inquire if an mxArray is of type double

**Fortran Syntax**

```
integer*4 function mxIsDouble(pm)
integer*4 pm
```

**Arguments**

pm
Pointer to an mxArray.

**Returns** 1 if true, 0 if false. 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.

**Purpose**

Inquire if an mxArray is full

**Fortran Syntax**

```
integer*4 function mxIsFull(pm)
integer*4 pm
```

**Arguments**

pm
Pointer to an mxArray.

**Returns**

1 if the mxArray is full, 0 if it is sparse.

**Description**

Call mxIsFull to determine if an mxArray is stored in full form or sparse form.

# mxIsNumeric

**Purpose** Inquire if an `mxArray` contains numeric data

**Fortran Syntax**

```
integer*4 function mxIsNumeric(pm)
integer*4 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 a `character` array.

**See Also** `mxIsString`

**Purpose** Inquire if an mxArray is sparse

**Fortran Syntax**

```
integer*4 function mxIsSparse(pm)
integer*4 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, mxGetJc, mxIsFull

# mxIsString

**Purpose** Inquire if an `mxArray` contains a `character` array

**Fortran Syntax**

```
integer*4 function mxIsString(pm)
integer*4 pm
```

**Arguments**

`pm`
Pointer to an `mxArray`.

**Returns** 1 if the `mxArray` contains a `character` array, and 0 otherwise.

**Description** Call `mxIsString` to inquire whether or not the `mxArray` contains a `character` array. The `DisplayMode` flag tells MATLAB whether to display the `mxArray` in numeric form or to interpret the elements as ASCII values and to display the `mxArray` as a `character` array, if the semicolon is omitted from a MATLAB statement.

Use `mxGetString` and `mxCreateString` to extract and insert `character` arrays into `mxArrays`.

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

**Purpose**

Set the ir array of a sparse mxArray

**Fortran Syntax**

```
subroutine mxSetIr(pm, ir)
integer*4 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.

**See Also**

mxCreateSparse, mxGetIr, mxGetJc, mxSetJc

# mxSetJc

**Purpose**

Set the `jc` array of a sparse `mxArray`

**Fortran Syntax**

```
subroutine mxSetJc(pm, jc)
integer*4 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`.

**See Also**

`mxGetIr`, `mxGetJc`, `mxSetIr`

**Purpose**

Set the number of rows

**Fortran Syntax**

```
subroutine mxSetM(pm, m)
integer*4 pm, 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`, `mxGetN`, `mxSetN`

# mxSetN

**Purpose** Set the number of columns

**Fortran Syntax**

```
subroutine mxSetN(pm, n)
integer*4 pm, 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`, `mxGetN`, `mxSetM`

**Purpose**

Set the name of an `mxArray`

**Fortran Syntax**

```
subroutine mxSetName(pm, name)
integer*4 pm
character*(32) name
```

**Arguments**

`pm`
Pointer to an `mxArray`.

`name`
The name you are assigning to the `mxArray`. The specified `name` can be up to 31 characters. If you specify a name longer than 31 characters, `mxSetName` assigns only the first 31 characters to the name.

**Description**

Call `mxSetName` to establish a name for an `mxArray` or to change an existing name.

`mxSetName` assigns the characters in `name` to a fixed-width section of memory. Do not deallocate this memory.

**See Also**

`mxGetName`

# mxSetNzmax

**Purpose**
Set the storage space for nonzero elements

**Fortran Syntax**

```
subroutine mxSetNzmax(pm, nzmax)
integer*4 pm, 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`

**Purpose** Set new imaginary data for an `mxArray`

**Fortran Syntax**

```
subroutine mxSetPi(pm, pi)
integer*4 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` to allocate this dynamic memory.

**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 `COMPLEX = 1` or by setting `pi` to a nonzero 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.

**See Also** `mxGetPi`, `mxGetPr`, `mxSetPr`

# mxSetPr

**Purpose**

Set new real data for an `mxArray`

**Fortran Syntax**

```
subroutine mxSetPr(pm, pr)
integer*4 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` to allocate this dynamic memory.

**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.

**See Also**

`mxGetPr`, `mxGetPi`, `mxSetPi`

10

# DDE Functions

| | |
|---|---|
| `ddeadv` | Set up advisory link between MATLAB and DDE server application |
| `ddeexec` | Send execution string to DDE server application |
| `ddeinit` | Initiate DDE conversation between MATLAB and another application |
| `ddepoke` | Send data from MATLAB to DDE server application |
| `ddereq` | Request data from DDE server application |
| `ddeterm` | Terminate DDE conversation between MATLAB and server application |
| `ddeunadv` | Release advisory link between MATLAB and DDE server application |

**Purpose**

Set up advisory link between MATLAB and DDE server application

**Syntax**

```
rc = ddeadv(channel, item, callback, upmtx, format, timeout)
```

**Arguments**

`rc`
The return code: 0 indicates the function call failed, 1 indicates it succeeded.

`channel`
The channel assigned to the conversation, returned by `ddeinit`.

`item`
A string that specifies the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.

`callback`
A string that specifies the callback that is evaluated on update notification. Changing `item` at the server causes `callback` to get passed to the `eval` function to be evaluated.

`upmtx`
(optional) A string that specifies the name of a matrix that holds data sent with update notification. If `upmtx` is included, changing `item` at the server causes `upmtx` to be updated with the revised data.

Specifying an update matrix creates a *hot link*. Omitting `upmtx` or specifying it as an empty string, creates a *warm link*. If `upmtx` exists in the workspace, its contents get overwritten. If `upmtx` does not exist, it is created.

`format`
(optional) A two-element array that specifies the format of the data to be sent on update.

The first element specifies the Windows clipboard format to use for the data. MATLAB supports only Text format, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `NUMERIC` (the default, which corresponds to a value of 0) and `STRING` (which corresponds to a value of 1).

The default format array is [1 0].

`timeout`
(optional) A scalar that specifies the time-out limit for this operation. `timeout` is specified in milliseconds (1000 milliseconds = 1 second).

If advisory link is not established within `timeout` milliseconds, the function fails. The default value of `timeout` is three seconds.

**Description**

`ddeadv` sets up an advisory link between MATLAB and a server application.

When the data identified by the `item` argument changes, the string specified by the `callback` argument is passed to the `eval` function and evaluated. If the advisory link is a hot link, DDE modifies `upmtx`, the update matrix, to reflect the data in `item`.

If `item` corresponds to a range of data values, a change to any value in the range causes `callback` to be evaluated.

**Example**

```
% Set up a hot link between a range of cells in Excel
% and the matrix 'x'.
% If successful, display the matrix.
rc = ddeadv(channel, 'r1c1:r5c5', 'disp(x)', 'x');
```

**Purpose** Send execution string to DDE server application

**Syntax**

```
rc = ddeexec(channel, command, item, timeout)
```

**Arguments**

`rc`
The return code: 0 indicates the function call failed, 1 indicates it succeeded.

`channel`
The channel assigned to the conversation, returned by `ddeinit`.

`command`
A string that specifies the command to be executed.

`item`
(optional) A string that specifies the DDE item name for execution. This argument is not used for many applications. If your application requires this argument, it provides additional information for `command`. Consult your server documentation for more information.

`timeout`
(optional) A scalar that specifies the time-out limit for this operation. `timeout` is specified in milliseconds (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

**Description** `ddeexec` sends a string for execution to another application via an established DDE conversation. Specify the string as the `command` argument.

**Example**

```
% Given the channel assigned to a conversation,
% send a command to Excel.
rc = ddeexec(channel, '[formula.goto("r1c1")]');
```

# ddeinit

**Purpose** Initiate DDE conversation between MATLAB and another application

**Syntax**

```
channel = ddeinit(service, topic)
```

**Arguments**

`channel`
The channel assigned to the conversation.

`service`
A string that specifies the service or application name for the conversation.

`topic`
A string that specifies the topic for the conversation.

**Description**

`ddeinit` requires two arguments: a service or application name and a topic for that service. The function returns a channel handle, which is used with other MATLAB DDE functions.

For more information about services and topics, see DDE Concepts and Terminology.

**Example**

```
% Initiate a conversation with Microsoft Excel
% for the spreadsheet 'forecast.xls'.
channel = ddeinit('excel', 'forecast.xls');
```

**Purpose** Send data from MATLAB to DDE server application

**Syntax**

```
rc = ddepoke(channel, item, data, format, timeout)
```

**Arguments**

`rc`
The return code: 0 indicates the function call failed, 1 indicates it succeeded.

`channel`
The channel assigned to the conversation, returned by `ddeinit`.

`item`
A string that specifies the DDE item for the data sent. `item` is the server data entity that is to contain the data sent in the `data` argument.

`data`
A matrix that contains the data to be sent.

`format`
(optional) A scalar that specifies the Windows clipboard format of the data. MATLAB supports only Text format, which corresponds to a value of 1.

`timeout`
(optional) A scalar that specifies the time-out limit for this operation. `timeout` is specified in milliseconds (1000 milliseconds = 1 second). The default `timeout` is three seconds.

**Description** `ddepoke` sends data to an application via an established DDE conversation. `ddepoke` formats the data matrix as follows before sending it to the server application:

- String matrices are converted, element by element, to characters and the resulting character buffer is sent.
- Numeric matrices are sent as tab-delimited columns and carriage-return, line-feed delimited rows of numbers. Only the real part of non-sparse matrices are sent.

**Example**

```
% Send a 5-by-5 identity matrix to Excel.
rc = ddepoke(channel, 'r1c1:r5c5', eye(5));
```

# ddereq

**Purpose** Request data from DDE server application

**Syntax**

```
data = ddereq(channel, item, format, timeout)
```

**Arguments**

`data`
A matrix that contains the requested data, empty if the function call failed.

`channel`
The channel assigned to the conversation, returned by `ddeinit`.

`item`
A string that specifies the server application's DDE item name for the data requested.

`format`
(optional) A two-element array that specifies the format of the data requested.

The first element indicates a Windows clipboard format to use for the request. MATLAB supports only Text format, which corresponds to a value of 1.

The second element of the format array specifies the type of the resultant matrix. The valid types are `NUMERIC` (the default, corresponding to a value of 0) and `STRING` (corresponding to a value of 1).

The default format array is [1 0].

`timeout`
(optional) A scalar that specifies the time-out limit for this operation. `timeout` is specified in milliseconds (1000 milliseconds = 1 second). The default `timeout` is three seconds.

**Description** `ddereq` requests data from a server application via an established DDE conversation. `ddereq` returns a matrix containing the requested data or an empty matrix if the function is unsuccessful.

**Example**

```
% Request a matrix of cells from Excel.
mymtx = ddereq(channel, 'r1c1:r10c10');
```

**Purpose** Terminate DDE conversation between MATLAB and server application

**Syntax**

```
rc = ddeterm(channel)
```

**Arguments**

`rc`
The return code: 0 indicates the function call failed, 1 indicates it succeeded.

`channel`
The channel assigned to the conversation, returned by `ddeinit`.

**Description** `ddeterm` takes one argument, the channel handle returned by the previous call to `ddeinit` that established the DDE conversation.

**Example**

```
% Terminate the DDE conversation.
rc = ddeterm(channel);
```

# ddeunadv

**Purpose**

Release an advisory link between MATLAB and DDE server application

**Syntax**

```
rc = ddeunadv(channel, item, format, timeout)
```

**Arguments**

`rc`
The return code: 0 indicates the function call failed, 1 indicates it succeeded.

`channel`
The channel assigned to the conversation, returned by `ddeinit`.

`item`
A string that specifies the DDE item name associated with the advisory link.

`format`
(optional) A two-element array that specifies the format of the data for the advisory link. If you specified a `format` argument on the `ddeadv` function call that defined the advisory link, you must specify the same value on the `ddeunadv` function call. See `ddeadv` for a description of the `format` array.

`timeout`
(optional) A scalar that specifies the time-out limit for this operation. `timeout` is specified in milliseconds (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

**Description**

`ddeunadv` releases the advisory link between MATLAB and the server application, established by an earlier `ddeadv` call. The `channel`, `item`, and `format` must be the same as those specified in the call to `ddeadv` that initiated the link. If you include the `timeout` argument but accept the default `format`, you must specify `format` as an empty matrix.

**Example**

```
% Release the hot link established in the ddeadv example.
rc = ddeunadv(channel, 'r1c1:r5c5');

% Release a hot link with default format and a timeout value.
rc = ddeunadv(chan, 'r1c1:r5c5', [], 6000);
```