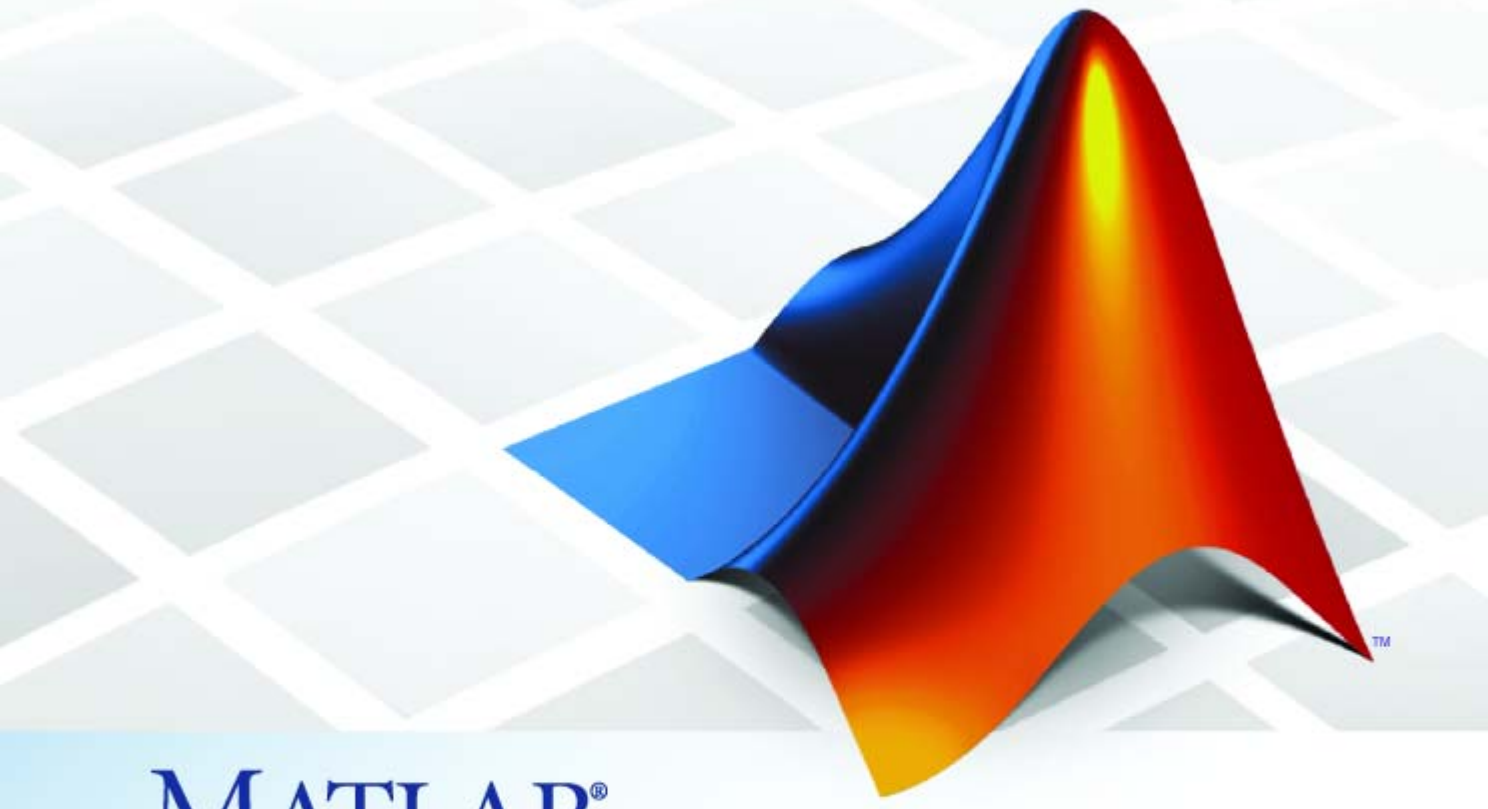
MATLAB® 7
C/C++ and Fortran API Reference



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB® C/C++ and Fortran API Reference

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

December 1996	First Printing	New for MATLAB 5 (Release 8)
May 1997	Online only	Revised for MATLAB 5.1 (Release 9)
January 1998	Online Only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online Only	Revised for MATLAB 5.3 (Release 11)
September 2000	Online Only	Revised for MATLAB 6.0 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
January 2003	Online only	Revised for MATLAB 6.5.1 (Release 13SP1)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised and renamed for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised and renamed for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised and renamed for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised and renamed for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised and renamed for MATLAB 7.10 (Release 2010a)

Contents

1

API Reference

MAT-File Library (p. 1-2)	Incorporate and use MATLAB® data in C/C++ and Fortran programs
MX Matrix Library (p. 1-2)	Create and manipulate MATLAB arrays from C/C++ and Fortran MEX and engine routines
MEX Library (p. 1-10)	Perform operations in MATLAB environment from C/C++ and Fortran MEX-files
Engine Library (p. 1-11)	Call MATLAB software from C/C++ and Fortran programs

See also “External Interfaces” in the MATLAB Function Reference for interfaces to shared libraries, the Sun™ Java™ programming language, Microsoft® .NET, Component Object Model (COM), and ActiveX® technologies, Web services, and serial port devices.

MAT-File Library

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

MX Matrix Library

mwIndex (C and Fortran)	Type for index values
mwPointer (Fortran)	Declare appropriate pointer type for platform
mwSignedIndex (C and Fortran)	Signed integer type for size values
mwSize (C and Fortran)	Type for size values
mxAddField (C and Fortran)	Add field to structure array
mxArray (C and Fortran)	Type for a MATLAB array

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

mxCopyPtrToComplex8 (Fortran)	Copy COMPLEX*8 values from pointer array to Fortran array
mxCopyPtrToInteger1 (Fortran)	Copy INTEGER*1 values from pointer array to Fortran array
mxCopyPtrToInteger2 (Fortran)	Copy INTEGER*2 values from pointer array to Fortran array
mxCopyPtrToInteger4 (Fortran)	Copy INTEGER*4 values from pointer array to Fortran array
mxCopyPtrToPtrArray (Fortran)	Copy pointer values from pointer array to Fortran array
mxCopyPtrToReal4 (Fortran)	Copy REAL*4 values from pointer array to Fortran array
mxCopyPtrToReal8 (Fortran)	Copy REAL*8 values from pointer array to Fortran array
mxCopyReal4ToPtr (Fortran)	Copy REAL*4 values from Fortran array to pointer array
mxCopyReal8ToPtr (Fortran)	Copy REAL*8 values from Fortran array to pointer array
mxCreateCellArray (C and Fortran)	Create unpopulated N-D cell mxArray
mxCreateCellMatrix (C and Fortran)	Create unpopulated 2-D cell mxArray
mxCreateCharArray (C and Fortran)	Create unpopulated N-D string mxArray
mxCreateCharMatrixFromStrings (C and Fortran)	Create populated 2-D string mxArray
mxCreateDoubleMatrix (C and Fortran)	Create 2-D, double-precision, floating-point mxArray initialized to 0
mxCreateDoubleScalar (C and Fortran)	Create scalar, double-precision array initialized to specified value
mxCreateLogicalArray (C)	Create N-D logical mxArray initialized to false

mxCreateLogicalMatrix (C)	Create 2-D, logical mxArray initialized to false
mxCreateLogicalScalar (C)	Create scalar, logical mxArray
mxCreateNumericArray (C and Fortran)	Create unpopulated N-D numeric mxArray
mxCreateNumericMatrix (C and Fortran)	Create numeric matrix initialized to 0
mxCreateSparse (C and Fortran)	Create 2-D unpopulated sparse mxArray
mxCreateSparseLogicalMatrix (C)	Create unpopulated 2-D, sparse, logical mxArray
mxCreateString (C and Fortran)	Create 1-by-N string mxArray initialized to specified string
mxCreateStructArray (C and Fortran)	Create unpopulated N-D structure mxArray
mxCreateStructMatrix (C and Fortran)	Create unpopulated 2-D structure mxArray
mxDestroyArray (C and Fortran)	Free dynamic memory allocated by mxCreate* functions
mxDuplicateArray (C and Fortran)	Make deep copy of array
mxFree (C and Fortran)	Free dynamic memory allocated by mxCalloc, mxMalloc, or mxRealloc
mxGetCell (C and Fortran)	Contents of mxArray cell
mxGetChars (C)	Pointer to character array data
mxGetClassID (C and Fortran)	Class of mxArray
mxGetClassName (C and Fortran)	Class of mxArray as string
mxGetData (C and Fortran)	Pointer to real data
mxGetDimensions (C and Fortran)	Pointer to dimensions array

mxGetElementSize (C and Fortran)	Number of bytes required to store each data element
mxGetEps (C and Fortran)	Value of eps
mxGetField (C and Fortran)	Field value, given field name and index, into structure array
mxGetFieldByNumber (C and Fortran)	Field value, given field number and index, into structure array
mxGetFieldNameByNumber (C and Fortran)	Field name, given field number, in structure array
mxGetFieldNumber (C and Fortran)	Field number, given field name, in structure array
mxGetImagData (C and Fortran)	Pointer to imaginary data of mxArray
mxGetInf (C and Fortran)	Value of infinity
mxGetIr (C and Fortran)	ir array of sparse matrix
mxGetJc (C and Fortran)	jc array of sparse matrix
mxGetLogicals (C)	Pointer to logical array data
mxGetM (C and Fortran)	Number of rows in mxArray
mxGetN (C and Fortran)	Number of columns in mxArray
mxGetNaN (C and Fortran)	Value of NaN (Not-a-Number)
mxGetNumberOfDimensions (C and Fortran)	Number of dimensions in mxArray
mxGetNumberOfElements (C and Fortran)	Number of elements in mxArray
mxGetNumberOfFields (C and Fortran)	Number of fields in structure mxArray
mxGetNzmax (C and Fortran)	Number of elements in ir, pr, and pi arrays
mxGetPi (C and Fortran)	Imaginary data elements in mxArray of type double
mxGetPr (C and Fortran)	Real data elements in mxArray of type double

mxGetProperty (C and Fortran)	Value of public property of MATLAB object
mxGetScalar (C and Fortran)	Real component of first data element in mxArray
mxGetString (C and Fortran)	Copy string mxArray to C-style string
mxIsCell (C and Fortran)	Determine whether input is cell mxArray
mxIsChar (C and Fortran)	Determine whether input is string mxArray
mxIsClass (C and Fortran)	Determine whether mxArray is member of specified class
mxIsComplex (C and Fortran)	Determine whether data is complex
mxIsDouble (C and Fortran)	Determine whether mxArray represents data as double-precision, floating-point numbers
mxIsEmpty (C and Fortran)	Determine whether mxArray is empty
mxIsFinite (C and Fortran)	Determine whether input is finite
mxIsFromGlobalWS (C and Fortran)	Determine whether mxArray was copied from MATLAB global workspace
mxIsInf (C and Fortran)	Determine whether input is infinite
mxIsInt16 (C and Fortran)	Determine whether mxArray represents data as signed 16-bit integers
mxIsInt32 (C and Fortran)	Determine whether mxArray represents data as signed 32-bit integers
mxIsInt64 (C and Fortran)	Determine whether mxArray represents data as signed 64-bit integers

`mxIsInt8 (C and Fortran)`	Determine whether `mxArray` represents data as signed 8-bit integers
`mxIsLogical (C and Fortran)`	Determine whether `mxArray` is of type `mxLogical`
`mxIsLogicalScalar (C)`	Determine whether scalar `mxArray` is of type `mxLogical`
`mxIsLogicalScalarTrue (C)`	Determine whether scalar `mxArray` of type `mxLogical` is `true`
`mxIsNaN (C and Fortran)`	Determine whether input is `NaN` (Not-a-Number)
`mxIsNumeric (C and Fortran)`	Determine whether `mxArray` is numeric
`mxIsSingle (C and Fortran)`	Determine whether `mxArray` represents data as single-precision, floating-point numbers
`mxIsSparse (C and Fortran)`	Determine whether input is sparse `mxArray`
`mxIsStruct (C and Fortran)`	Determine whether input is structure `mxArray`
`mxIsUint16 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 16-bit integers
`mxIsUint32 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 32-bit integers
`mxIsUint64 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 64-bit integers
`mxIsUint8 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 8-bit integers
`mxLogical (C)`	Type for logical `mxArray`

`mxMalloc (C and Fortran)`	Allocate dynamic memory using MATLAB memory manager
`mxRealloc (C and Fortran)`	Reallocate memory
`mxRemoveField (C and Fortran)`	Remove field from structure array
`mxSetCell (C and Fortran)`	Set value of one cell of `mxArray`
`mxSetClassName (C)`	Convert structure array to MATLAB object array
`mxSetData (C and Fortran)`	Set pointer to data
`mxSetDimensions (C and Fortran)`	Modify number of dimensions and size of each dimension
`mxSetField (C and Fortran)`	Set structure array field, given structure field name and array index
`mxSetFieldByNumber (C and Fortran)`	Set structure array field, given field number and index
`mxSetImagData (C and Fortran)`	Set imaginary data pointer for `mxArray`
`mxSetIr (C and Fortran)`	Set `ir` array of sparse `mxArray`
`mxSetJc (C and Fortran)`	Set `jc` array of sparse `mxArray`
`mxSetM (C and Fortran)`	Set number of rows in `mxArray`
`mxSetN (C and Fortran)`	Set number of columns in `mxArray`
`mxSetNzmax (C and Fortran)`	Set storage space for nonzero elements
`mxSetPi (C and Fortran)`	Set new imaginary data for `mxArray`
`mxSetPr (C and Fortran)`	Set new real data for `mxArray`
`mxSetProperty (C and Fortran)`	Set value of public property of MATLAB object

MEX Library

`mexAtExit (C and Fortran)`	Register function to call when MEX-function cleared or MATLAB software terminates
`mexCallMATLAB (C and Fortran)`	Call MATLAB function, user-defined function, or MEX-file
`mexCallMATLABWithTrap (C and Fortran)`	Call MATLAB function, user-defined function, or MEX-file and capture error information
`mexErrMsgIdAndTxt (C and Fortran)`	Display error message with identifier and return to MATLAB prompt
`mexErrMsgTxt (C and Fortran)`	Display error message and return to MATLAB prompt
`mexEvalString (C and Fortran)`	Execute MATLAB command in caller workspace
`mexEvalStringWithTrap (C and Fortran)`	Execute MATLAB command in caller workspace and capture error information
`mexFunction (C and Fortran)`	Entry point to C/C++ or Fortran MEX-file
`mexFunctionName (C and Fortran)`	Name of current MEX-function
`mexGet (C)`	Value of specified Handle Graphics® property
`mexGetVariable (C and Fortran)`	Copy of variable from specified workspace
`mexGetVariablePtr (C and Fortran)`	Read-only pointer to variable from another workspace
`mexIsGlobal (C and Fortran)`	Determine whether `mxArray` has global scope
`mexIsLocked (C and Fortran)`	Determine whether MEX-file is locked

`mexLock (C and Fortran)`	Prevent clearing MEX-file from memory
`mexMakeArrayPersistent (C and Fortran)`	Make `mxArray` persist after MEX-file completes
`mexMakeMemoryPersistent (C and Fortran)`	Make memory allocated by MATLAB software persist after MEX-function completes
`mexPrintf (C and Fortran)`	ANSI® C `printf`-style output routine
`mexPutVariable (C and Fortran)`	Copy `mxArray` from MEX-function into specified workspace
`mexSet (C)`	Set value of specified Handle Graphics property
`mexSetTrapFlag (C and Fortran)`	Control response of `mexCallMATLAB` to errors
`mexUnlock (C and Fortran)`	Allow clearing MEX-file from memory
`mexWarnMsgIdAndTxt (C and Fortran)`	Display warning message with identifier
`mexWarnMsgTxt (C and Fortran)`	Warning message

Engine Library

`engClose (C and Fortran)`	Quit MATLAB engine session
`engEvalString (C and Fortran)`	Evaluate expression in string
`engGetVariable (C and Fortran)`	Copy variable from MATLAB engine workspace
`engGetVisible (C)`	Determine visibility of MATLAB engine session
`Engine (C)`	Type for a MATLAB engine
`engOpen (C and Fortran)`	Start MATLAB engine session

`engOpenSingleUse (C)`	Start MATLAB engine session for single, nonshared use
`engOutputBuffer (C and Fortran)`	Specify buffer for MATLAB output
`engPutVariable (C and Fortran)`	Put variable into MATLAB engine workspace
`engSetVisible (C)`	Show or hide MATLAB engine session

2

API Reference

engClose (C and Fortran)

Purpose

Quit MATLAB engine session

C Syntax

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

Fortran Syntax

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

Arguments

`ep`

Engine pointer

Returns

`0` on success, and `1` otherwise. Possible failure includes attempting to terminate an already-terminatedMATLAB engine session.

Description

This routine sends a quit command to the MATLAB engine session and closes the connection.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `engdemo.c` for a C example on UNIX® operating systems.
- `engwindemo.c` for a C example on Microsoft Windows® operating systems.
- `fengdemo.F` for a Fortran example.

See Also

`engOpen`

Purpose

Evaluate expression in string

C Syntax

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

Fortran Syntax

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

Arguments

`ep`
Engine pointer

`string`
String to execute

Returns

`0` if the MATLAB engine session evaluated the command, and a nonzero value if unsuccessful. Possible reasons for failure include the engine session is no longer running or the engine pointer is invalid or `NULL`.

Error Handling

If `string` detects an error, MATLAB terminates and returns control to the MATLAB prompt.

Description

`engEvalString` evaluates the expression contained in `string` for the MATLAB engine session, `ep`, previously started by `engOpen`.

UNIX Operating Systems

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

To turn off output buffering in C, use:

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

To turn off output buffering in Fortran, use:

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

Microsoft Windows Operating Systems

On a Windows system, `engEvalString` communicates with MATLAB software using a Component Object Model (COM) interface.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.
- `fengdemo.F` for a Fortran example.

See Also

`engOpen`, `engOutputBuffer`

Purpose

Copy variable from MATLAB engine workspace

C Syntax

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

Fortran Syntax

```
mwPointer engGetVariable(ep, name)
mwPointer ep
character*(*) name
```

Arguments

`ep`
: Engine pointer

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

Returns

Pointer to a newly allocated `mxArray` structure, or `NULL` if the attempt fails. `engGetVariable` fails if the named variable does not exist.

Description

`engGetVariable` reads the named `mxArray` from the MATLAB engine session associated with `ep`.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.

See Also

`engPutVariable`, `mxDestroyArray`

engGetVisible (C)

Purpose

Determine visibility of MATLAB engine session

C Syntax

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

Arguments

`ep`
Engine pointer

`value`
Pointer to value returned from `engGetVisible`

Returns

Microsoft Windows Operating Systems Only

`0` on success, and `1` otherwise.

Description

`engGetVisible` returns the current visibility setting for MATLAB engine session, `ep`. A *visible* engine session runs in a window on the Windows desktop, thus making the engine available for user interaction. MATLAB removes an invisible session from the desktop.

Examples

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

```
Engine *ep;
bool vis;

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

To determine the current visibility setting, use:

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

See Also

`engSetVisible`

Purpose

Type for a MATLAB engine

Description

A handle to a MATLAB engine object.

`Engine` is a C language opaque type.

You can call MATLAB software as a computational engine by writing C and Fortran programs that use the MATLAB engine library, described in "Engine Library" on page 1-11. `Engine` is the link between your program and the separate MATLAB engine process.

The header file containing this type is:

```
#include "engine.h"
```

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `engdemo.c` shows how to call the MATLAB engine functions from a C program.
- `engwindemo.c` show how to call the MATLAB engine functions from a C program for Windows systems.
- `fengdemo.F` shows how to call the MATLAB engine functions from a Fortran program.

See Also

`engOpen`

engOpen (C and Fortran)

Purpose

Start MATLAB engine session

C Syntax

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

Fortran Syntax

```
mwPointer engOpen(startcmd)
character*(*) startcmd
```

Arguments

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

Returns

Pointer to an engine handle, or `NULL` if the open fails.

Description

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

`engOpen` starts a MATLAB process using the command specified in the string `startcmd`, establishes a connection, and returns an engine pointer.

On UNIX systems, if `startcmd` is `NULL` or the empty string, `engOpen` starts a MATLAB process on the current host using the command `matlab`. If `startcmd` is a `hostname`, `engOpen` starts a MATLAB process 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), MATLAB executes the string literally.

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

1. Creates two pipes.

2. Forks a new process. Sets up the pipes to pass *stdin* and *stdout* from MATLAB (parent) software to two file descriptors in the engine program (child).

3. Executes a command to run MATLAB software (`rsh` for remote execution).

On Windows systems, `engOpen` opens a COM channel to MATLAB. The MATLAB software you registered during installation starts. If you did not register during installation, on the command line you can enter the command:

```
matlab /regserver
```

See "Introducing MATLAB COM Integration" for additional details.

Examples

See the following examples in *matlabroot*`/extern/examples/eng_mat`.

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.
- `fengdemo.F` for a Fortran example.

engOpenSingleUse (C)

Purpose

Start MATLAB engine session for single, nonshared use

C Syntax

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

Arguments

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

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

`retstatus`
: Return status; possible cause of failure.

Returns

Microsoft Windows Operating Systems Only

Pointer to an engine handle, or `NULL` if the open fails.

UNIX Operating Systems

Not supported on UNIX systems.

Description

This routine allows you to start multiple MATLAB processes 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. Each call to `engOpenSingleUse` starts a new MATLAB process.

`engOpenSingleUse` opens a COM channel to MATLAB. This starts the MATLAB software you 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 engine server. `engOpenSingleUse` differs from `engOpen`, which allows multiple applications to use the same engine server.

See “Introducing MATLAB COM Integration” for additional details.

engOutputBuffer (C and Fortran)

Purpose

Specify buffer for MATLAB output

C Syntax

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

Fortran Syntax

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

Arguments

`ep`
: Engine pointer

`p`
: Pointer to character buffer

`n`
: Length of buffer `p`

Returns

`1` if you pass it a NULL engine pointer. Otherwise, returns `0`.

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. A call to `engOutputBuffer` with a buffer of nonzero length tells any subsequent calls to `engEvalString` to save output in the character buffer pointed to by `p`.

To turn off output buffering in C, use:

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

To turn off output buffering in Fortran, use:

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

Note The buffer returned by `engEvalString` is not `NULL` terminated.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.
- `fengdemo.F` for a Fortran example.

See Also

`engOpen`, `engEvalString`

engPutVariable (C and Fortran)

Purpose

Put variable into MATLAB engine workspace

C Syntax

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

Fortran Syntax

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

Arguments

`ep`
: Engine pointer

`name`
: Name of `mxArray` in the engine workspace

`pm`
: `mxArray` pointer

Returns

`0` if successful and `1` if an error occurs.

Description

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

If the `mxArray` does not exist in the workspace, the function creates it. If an `mxArray` with the same name exists in the workspace, the function replaces the existing `mxArray` with the new `mxArray`.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

The engine application owns the original `mxArray` and is responsible for freeing its memory. Although the `engPutVariable` function sends a copy of the `mxArray` to the MATLAB workspace, the engine application does not need to account for or free memory for the copy.

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- engdemo.c for a C example on UNIX operating systems.
- engwindemo.c for a C example on Microsoft Windows operating systems.

See Also

engGetVariable

engSetVisible (C)

Purpose

Show or hide MATLAB engine session

C Syntax

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

Arguments

`ep`
: Engine pointer

`value`
: Value to set the `Visible` property to. Set `value` to `1` to make the engine window visible, or to `0` to make it invisible.

Returns

Microsoft Windows Operating Systems Only

`0` on success, and `1` otherwise.

Description

`engSetVisible` makes the window for the MATLAB engine session, `ep`, either visible or invisible on the Windows desktop. You can use this function to enable or disable user interaction with the MATLAB engine session.

Examples

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

```
Engine *ep;
bool vis;

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

To determine the current visibility setting, use:

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

See Also

`engGetVisible`

Purpose

Close MAT-file

C Syntax

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

Fortran Syntax

```
integer*4 matClose(mfp)
mwPointer mfp
```

Arguments

`mfp`
Pointer to MAT-file information

Returns

`EOF` in C (`-1` in Fortran) for a write error, and `0` if successful.

Description

`matClose` closes the MAT-file associated with `mfp`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`
- `matdemo1.F`
- `matdemo2.F`

See Also

`matOpen`

matDeleteVariable (C and Fortran)

Purpose

Delete named `mxArray` from MAT-file

C Syntax

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

Fortran Syntax

```
integer*4 matDeleteVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Name of `mxArray` to delete

Returns

`0` if successful, and nonzero otherwise.

Description

`matDeleteVariable` deletes the named `mxArray` from the MAT-file pointed to by `mfp`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`

Purpose

Type for a MAT-file

Description

A handle to a MAT-file object. A MAT-file is the data file format MATLAB software uses for saving data to your disk.

`MATFile` is a C language opaque type.

The MAT-file interface library contains routines for reading and writing MAT-files. For a list of these routines, see “MAT-File Library” on page 1-2 in the C/C++ and Fortran API Reference. Call these routines from your own C/C++ and Fortran programs, using `MATFile` to access your data file.

The header file containing this type is:

```
#include "mat.h"
```

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`
- `matdemo1.F`
- `matdemo2.F`

See Also

`matOpen`, `matClose`, `matPutVariable`, `matGetVariable`, `mxDestroyArray`

matGetDir (C and Fortran)

Purpose

Directory of mxArrays in MAT-file

C Syntax

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

Fortran Syntax

```
mwPointer matGetDir(mfp, num)
mwPointer mfp
integer*4 num
```

Arguments

mfp
: Pointer to MAT-file information

num
: Pointer to the variable containing the number of mxArrays in the MAT-file

Returns

Pointer to an internal array containing pointers to the names of the mxArrays in the MAT-file pointed to by mfp. In C, each name is a NULL-terminated string. The num output argument is the length of the internal array (number of mxArrays in the MAT-file). If num is zero, mfp contains no arrays.

matGetDir returns NULL in C (0 in Fortran). If matGetDir fails, sets num to a negative number.

Description

This routine provides you with a list of the names of the mxArrays contained within a MAT-file.

matGetDir allocates memory for the internal array of strings using a mxCalloc. You must free the memory using mxFree when you are finished with the array.

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

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matcreat.c

- matdgns.c
- matdemo2.F

matGetFp (C)

Purpose

File pointer to MAT-file

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information

Returns

C file handle to the MAT-file with handle `mfp`. Returns `NULL` if `mfp` is a handle to a MAT-file in HDF5-based format.

Description

Use `matGetFp` to obtain a C file handle to a MAT-file. Standard C library routines, like `ferror` and `feof`, use file handle to investigate errors.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`

Purpose

Read next `mxArray` from MAT-file

C Syntax

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

Fortran Syntax

```
mwPointer matGetNextVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Pointer to the variable containing the `mxArray` name

Returns

Pointer to a newly allocated `mxArray` structure representing the next `mxArray` from the MAT-file pointed to by `mfp`. The function returns the name of the `mxArray` in `name`.

`matGetNextVariable` returns `NULL` in C (`0` in Fortran) for end-of-file or if there is an error condition. In C, use `feof` and `ferror` from the Standard C Library to determine status.

Description

`matGetNextVariable` allows you to step sequentially through a MAT-file and read all the `mxArrays` in a single pass. The function reads and returns the next `mxArray` from the MAT-file pointed to by `mfp`.

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

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

The order of variables returned from successive calls to `matGetNextVariable` is not guaranteed to be the same order in which the variables were written.

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matdgns.c

See Also

matGetNextVariableInfo, matGetVariable, mxDestroyArray

Purpose

Load array header information only

C Syntax

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

Fortran Syntax

```
mwPointer matGetNextVariableInfo(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Pointer to the variable containing the `mxArray` name

Returns

Pointer to a newly allocated `mxArray` structure representing header information for the next `mxArray` from the MAT-file pointed to by `mfp`. The function returns the name of the `mxArray` in `name`.

`matGetNextVariableInfo` returns `NULL` in C (`0` in Fortran) when the end-of-file is reached or if there is an error condition. In C, use `feof` and `ferror` from the Standard C Library to determine status.

Description

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

If `pr`, `pi`, `ir`, and `jc` are nonzero values when loaded with `matGetVariable`, `matGetNextVariableInfo` sets them to `-1` instead. These headers are for informational use only. *Never* pass this data back to the MATLAB workspace or save it to MAT-files.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

The order of variables returned from successive calls to `matGetNextVariableInfo` is not guaranteed to be the same order in which the variables were written.

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matdgns.c

See Also

matGetNextVariable, matGetVariableInfo

Purpose

Read mxArray from MAT-files

C Syntax

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

Fortran Syntax

```
mwPointer matGetVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

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

Returns

Pointer to a newly allocated `mxArray` structure representing the `mxArray` named by `name` from the MAT-file pointed to by `mfp`.

`matGetVariable` returns `NULL` in C (`0` in Fortran) if the attempt to return the `mxArray` named by `name` fails.

Description

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

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`

See Also

`matPutVariable`, `mxDestroyArray`

matGetVariableInfo (C and Fortran)

Purpose

Load array header information only

C Syntax

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

Fortran Syntax

```
mwPointer matGetVariableInfo(mfp, name);
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

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

Returns

Pointer to a newly allocated `mxArray` structure representing header information for the `mxArray` named by `name` from the MAT-file pointed to by `mfp`.

`matGetVariableInfo` returns `NULL` in C (`0` in Fortran) if the attempt to return header information for the `mxArray` named by `name` fails.

Description

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

If `pr`, `pi`, `ir`, and `jc` are nonzero values when loaded with `matGetVariable`, `matGetVariableInfo` sets them to `-1` instead. These headers are for informational use only. *Never* pass this data back to the MATLAB workspace or save it to MAT-files.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`

- matdgns.c

See Also

matGetVariable

matOpen (C and Fortran)

Purpose

Open MAT-file

C Syntax

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

Fortran Syntax

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

Arguments

`filename`

Name of file to open

`mode`

File opening mode. The following table lists valid values for `mode`.

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

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

Returns

File handle, or `NULL` in C (`0` in Fortran) if the open fails.

Description

This routine opens a MAT-file for reading and writing.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`
- `matdemo1.F`
- `matdemo2.F`

See Also

`matClose`

matPutVariable (C and Fortran)

Purpose

Write mxArray to MAT-file

C Syntax

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

Fortran Syntax

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

Arguments

mfp
: Pointer to MAT-file information

name
: Name of mxArray to put into MAT-file

pm
: mxArray pointer

Returns

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

Description

This routine puts an mxArray into a MAT-file.

matPutVariable writes mxArray pm to the MAT-file mfp. If the mxArray does not exist in the MAT-file, the function appends it to the end. If an mxArray with the same name exists in the file, the function replaces the existing mxArray with the new mxArray by rewriting the file.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include i, j, mode, char, size, or path. To determine whether a particular name is associated with a MATLAB function, use the which function.

The size of the new mxArray can be different from the existing mxArray.

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matcreat.c
- matdgns.c

See Also

matGetVariable, matGetFp

matPutVariableAsGlobal (C and Fortran)

Purpose

Put mxArray into MAT-file as originating from global workspace

C Syntax

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

Fortran Syntax

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

Arguments

mfp
: Pointer to MAT-file information

name
: Name of mxArray to put into MAT-file

pm
: mxArray pointer

Returns

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

Description

This routine puts an mxArray into a MAT-file. matPutVariableAsGlobal is like matPutVariable, except that MATLAB software loads the array into the global workspace and sets a reference to it in the local workspace. If you write to a MATLAB 4 format file, matPutVariableAsGlobal does not load it as global and has the same effect as matPutVariable.

matPutVariableAsGlobal writes mxArray pm to the MAT-file mfp. If the mxArray does not exist in the MAT-file, the function appends it to the end. If an mxArray with the same name exists in the file, the function replaces the existing mxArray with the new mxArray by rewriting the file.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include i, j, mode,

`char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

The size of the new `mxArray` can be different from the existing `mxArray`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`

See Also

`matPutVariable`, `matGetFp`

mexAtExit (C and Fortran)

Purpose

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

C Syntax

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

Fortran Syntax

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

Arguments

`ExitFcn`

Pointer to function you want to run on exit

Returns

Always returns `0`.

Description

Use `mexAtExit` to register a function to call just before clearing the MEX-function or terminating MATLAB. `mexAtExit` gives your MEX-function a chance to perform tasks such as freeing persistent memory and closing files. Other typical tasks include closing streams or sockets.

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

If a MEX-function is locked, you cannot clear the MEX-file. Consequently, if you attempt to clear a locked MEX-file, MATLAB does not call the `ExitFcn`.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexatexit.c`

See Also

`mexLock`, `mexUnlock`, `mexSetTrapFlag`

Purpose

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

C Syntax

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

Fortran Syntax

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

Arguments

`nlhs`
: Number of output arguments

`plhs`
: Array of pointers to output arguments

`nrhs`
: Number of input arguments

`prhs`
: Array of pointers to input arguments

`functionName`
: Character string containing name of the MATLAB built-in function, operator, user-defined function, or MEX-file you are calling

Returns

`0` if successful, and a nonzero value if unsuccessful.

Description

Call `mexCallMATLAB` to invoke internal MATLAB numeric functions, MATLAB operators, user-defined functions, or other MEX-files. Both `mexCallMATLAB` and `mexEvalString` execute MATLAB commands. Use `mexCallMATLAB` for returning results (left-hand side arguments) back to the MEX-file. The `mexEvalString` function cannot return values to the MEX-file.

For a complete description of the input and output arguments passed to `functionName`, see `mexFunction`. When calling the `mexCallMATLAB`

function, the number of output arguments `nlhs` and input arguments `nrhs` must be less than or equal to 50.

MATLAB allocates dynamic memory to store the `mxArrays` in `plhs`. MATLAB automatically deallocates the dynamic memory when you clear the MEX-file. However, if heap space is at a premium, call `mxDestroyArray` when you are finished with the `mxArrays` `plhs` points to.

If `functionName` is an operator, place the operator inside a pair of single quotes, for example, `'+'`.

Note It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`.

This function returns two variables but only assigns one of them a value:

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

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is now type `mxUNKNOWN_CLASS`.

Error Handling

If `functionName` detects an error, MATLAB terminates the MEX-file and returns control to the MATLAB prompt. If you want to trap errors, use the `mexCallMATLABWithTrap` function.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexcallmatlab.c`
- `mexevalstring.c`
- `mexsettrapflag.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `sincall.c`
- `sincall.F`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`
- `mxisclass.c`

See Also

`mexFunction`, `mexCallMATLABWithTrap`, `mexEvalString`, `mxDestroyArray`

mexCallMATLABWithTrap (C and Fortran)

Purpose

Call MATLAB function, user-defined function, or MEX-file and capture error information

C Syntax

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

Fortran Syntax

```
mwPointer mexCallMATLABWithTrap(nlhs, plhs, nrhs, prhs, functionName)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) functionName
```

Arguments

For more information about arguments, see `mexCallMATLAB`.

`nlhs`
: Number of desired output arguments.

`plhs`
: Array of pointers to output arguments.

`nrhs`
: Number of input arguments.

`prhs`
: Array of pointers to input arguments.

`functionName`
: Character string containing the name of the MATLAB built-in function, operator, function, or MEX-file that you are calling.

Returns

`NULL` if no error occurred; otherwise, a pointer to an `mxArray` of class `MException`.

Description

The `mexCallMATLABWithTrap` function performs the same function as `mexCallMATLAB`. However, if MATLAB detects an error when executing `functionName`, MATLAB returns control to the line in the MEX-file immediately following the call to `mexCallMATLABWithTrap`. For information about `MException`, see "Responding to an Exception"

See Also

mexCallMATLAB, MException

mexErrMsgIdAndTxt (C and Fortran)

Purpose

Display error message with identifier and return to MATLAB prompt

C Syntax

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

Fortran Syntax

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

Arguments

`errorid`
: String containing a MATLAB message identifier. For information on creating identifiers, see "Message Identifiers" in the MATLAB Programming Fundamentals documentation.

`errormsg`
: String to display. In C, the string can include conversion specifications, used by the ANSI C `printf` function.

`...`
: In C, any arguments used in the message. Each argument must have a corresponding conversion specification.

Description

The `mexErrMsgIdAndTxt` function writes an error message to the MATLAB window. For more information, see the `error` function syntax statement using a message identifier. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

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

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

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

Remarks

In addition to the `errorid` and `errormsg`, the `mexerrmsgtxt` function determines where the error occurred, and displays the following information. For example, in the function `foo`, `mexerrmsgtxt` displays:

```
??? Error using ==> foo
```

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillGetPr.c`
- `matrixDivide.c`
- `timestwo.F`
- `xtimesy.F`

See Also

`mexErrMsgTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

mexErrMsgTxt (C and Fortran)

Purpose

Display error message and return to MATLAB prompt

C Syntax

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

Fortran Syntax

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

Arguments

`errormsg`
: String containing the error message to display

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 might have a memory management compatibility problem. For more information, see "Memory Management Issues".

Remarks

In addition to the `errormsg`, the `mexerrmsgtxt` function determines where the error occurred, and displays the following information. If an error labeled `Print my error message` occurs in the function `foo`, `mexerrmsgtxt` displays:

```
??? Error using ==> foo
Print my error message
```

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- xtimesy.c
- convec.c
- findnz.c
- fulltosparse.c
- phonebook.c
- revord.c
- timestwo.c

See Also

mexErrMsgIdAndTxt, mexWarnMsgIdAndTxt, mexWarnMsgTxt

mexEvalString (C and Fortran)

Purpose

Execute MATLAB command in caller workspace

C Syntax

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

Fortran Syntax

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

Arguments

`command`
String containing 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. Use `mexCallMATLAB` for returning results (left-hand side arguments) back to the MEX-file. The `mexEvalString` function cannot return values to the MEX-file.

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

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function. For an example about conflicts between function and variable names, see "Naming Variables" in MATLAB Programming Fundamentals documentation.

Error Handling

If `command` detects an error, MATLAB terminates the MEX-file and returns control to the MATLAB prompt. If you want to trap errors, use the `mexEvalStringWithTrap` function.

Examples

See the following examples in *matlabroot*/extern/examples/mex.

- mexevalstring.c

See Also

mexCallMATLAB, mexEvalStringWithTrap

mexEvalStringWithTrap (C and Fortran)

Purpose

Execute MATLAB command in caller workspace and capture error information

C Syntax

```
#include "mex.h"
mxArray *mexEvalStringWithTrap(const char *command);
```

Fortran Syntax

```
mwPointer mexEvalStringWithTrap(command)
character*(*) command
```

Arguments

`command`
String containing the MATLAB command to execute

Returns

Object `ME` of class `MException`

Description

The `mexEvalStringWithTrap` function performs the same function as `mexEvalString`. However, if MATLAB detects an error when executing `command`, MATLAB returns control to the line in the MEX-file immediately following the call to `mexEvalStringWithTrap`.

See Also

`mexEvalString`, `MException`, `mexCallMATLAB`

Purpose

Entry point to C/C++ or Fortran MEX-file

C Syntax

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

Fortran Syntax

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

Arguments

`nlhs`
: Number of expected output `mxArray`s

`plhs`
: Array of pointers to the expected output `mxArray`s

`nrhs`
: Number of input `mxArray`s

`prhs`
: Array of pointers to the input `mxArray`s. Do not modify any `prhs` values in your MEX-file. Changing the data in these read-only `mxArray`s can produce undesired side effects.

Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of the gateway function in C (subroutine in Fortran) which every MEX-file requires. When you invoke a MEX-function, MATLAB software 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. MATLAB displays an error message if it cannot find a routine named `mexFunction` inside the MEX-file.

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. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is an array whose length is `nlhs`, where your function must set pointers for the returned left-hand side `mxArray`s.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexfunction.c`

Purpose

Name of current MEX-function

C Syntax

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

Fortran Syntax

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

Returns

Name of the current MEX-function.

Description

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexgetarray.c`

mexGet (C)

Purpose

Value of specified Handle Graphics property

C Syntax

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

Arguments

`handle`
: Handle to a particular graphics object

`property`
: Handle Graphics property

Returns

Value of the specified property in the specified graphics object on success. Returns `NULL` on failure. Do not modify the return argument from `mexGet`. Changing the data in a `const` (read-only) `mxArray` can produce undesired side effects.

Description

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexget.c`

See Also

`mexSet`

Purpose

Copy of variable from specified workspace

C Syntax

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

Fortran Syntax

```
mwPointer mexGetVariable(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`

Specifies where `mexGetVariable` searches for array `varname`. The possible values are:

`base`	Search for the variable in the base workspace.
`caller`	Search for the variable in the caller workspace.
`global`	Search for the variable in the global workspace.

`varname`

Name of the variable to copy

Returns

Copy of the variable on success. Returns `NULL` in C (`0` on Fortran) on failure. A common cause of failure is specifying a variable that is not currently in the workspace. Perhaps the variable was in the workspace at one time but has since been cleared.

Description

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

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

Examples

See the following examples in *matlabroot*/extern/examples/mex.

- mexgetarray.c

See Also

mexGetVariablePtr, mexPutVariable, mxDestroyArray

Purpose

Read-only pointer to variable from another workspace

C Syntax

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

Fortran Syntax

```
mwPointer mexGetVariablePtr(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`

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

`base`	Search for the variable in the base workspace.
`caller`	Search for the variable in the caller workspace.
`global`	Search for the variable in the global workspace.

`varname`

Name of a variable in another workspace. This is a variable name, not an `mxArray` pointer.

Returns

Read-only pointer to the `mxArray` on success. Returns `NULL` in C (`0` in Fortran) on failure.

Description

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

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxislogical.c

See Also

mexGetVariable

Purpose

Determine whether mxArray has global scope

C Syntax

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

Fortran Syntax

```
integer*4 mexIsGlobal(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

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

Description

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxislogical.c

See Also

mexGetVariable, mexGetVariablePtr, mexPutVariable, global

mexIsLocked (C and Fortran)

Purpose

Determine whether MEX-file is locked

C Syntax

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

Fortran Syntax

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

Returns

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

Description

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexlock.c`
- `mexlockf.F`

See Also

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

Purpose

Prevent clearing MEX-file from memory

C Syntax

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

Fortran Syntax

```
subroutine mexLock()
```

Description

By default, MEX-files are unlocked, meaning you can clear them at any time. Call `mexLock` to prohibit clearing a MEX-file.

To unlock a MEX-file, you must call `mexUnlock`. Do not use the `munlock` function.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexlock.c`
- `mexlockf.F`

See Also

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

mexMakeArrayPersistent (C and Fortran)

Purpose

Make mxArray persist after MEX-file completes

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray created by an mxCreate* function

Description

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

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

See Also

mexAtExit, mxDestroyArray, mexLock, mexMakeMemoryPersistent, and the mxCreate* functions

Purpose

Make memory allocated by MATLAB software persist after MEX-function completes

C Syntax

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

Fortran Syntax

```
subroutine mexMakeMemoryPersistent(ptr)
mwPointer ptr
```

Arguments

`ptr`

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

Description

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

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

See Also

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

mexPrintf (C and Fortran)

Purpose

ANSI C `printf`-style output routine

C Syntax

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

Fortran Syntax

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

Arguments

`message`
: String to display. In C, the string can include conversion specifications, used by the ANSI C `printf` function.

`...`
: In C, any arguments used in the message. Each argument must have a corresponding conversion specification.

Returns

Number of characters printed including characters specified with backslash codes, such as `\n` and `\b`.

Description

This routine prints a string on the screen and in the diary (if the diary is in use). It provides a callback to the standard C `printf` routine already linked inside MATLAB software, which avoids linking the entire `stdio` library into your MEX-file.

In a C MEX-file, you must call `mexPrintf` instead of `printf` to display a string.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexfunction.c`

See the following examples in *matlabroot*/extern/examples/refbook.

- phonebook.c

See Also

mexErrMsgIdAndTxt, mexErrMsgTxt, mexWarnMsgIdAndTxt, mexWarnMsgTxt

mexPutVariable (C and Fortran)

Purpose

Copy mxArray from MEX-function into specified workspace

C Syntax

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

Fortran Syntax

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

Arguments

`workspace`
Specifies scope of the array you are copying. Values for `workspace` are:

`base`	Copy `mxArray` to the base workspace.
`caller`	Copy `mxArray` to the caller workspace.
`global`	Copy `mxArray` to the list of global variables.

`varname`
Name of `mxArray` in the workspace

`pm`
Pointer to the `mxArray`

Returns

`0` on success; `1` on failure. A possible cause of failure is that `pm` is `NULL` in C (`0` in Fortran).

Description

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

`mexPutVariable` makes the array accessible to other entities, such as MATLAB, user-defined functions, or other MEX-functions.

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

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

```
Peaches
1     2     3     4
```

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

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

The value passed by `mexPutVariable` replaces the old value of `Peaches`.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexgetarray.c`

See Also

`mexGetVariable`

mexSet (C)

Purpose

Set value of specified Handle Graphics property

C Syntax

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

Arguments

`handle`
: Handle to a particular graphics object

`property`
: String naming a Handle Graphics property

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

Returns

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

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

Description

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

Examples

See the following examples in the *`matlabroot`*`/extern/examples/mex` folder.

- `mexcallmatlab.c`
- `mexget.c`
- `mexgetarray.c`

See Also

`mexGet`

Purpose

Control response of `mexCallMATLAB` to errors

C Syntax

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

Note The `mexsettrapflag` function will be removed in a future version of MATLAB software.

Fortran Syntax

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

Arguments

`trapflag`

Control flag. Possible values are:

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

Description

Call `mexSetTrapFlag` to control the MATLAB response to errors in `mexCallMATLAB`.

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

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

If you call `mexSetTrapFlag`, the value of the `trapflag` you set remains in effect until the next call to `mexSetTrapFlag` within that MEX-file or,

if there are no more calls to `mexSetTrapFlag`, until the MEX-file exits. If a routine defined in a MEX-file calls another MEX-file, MATLAB:

1. Saves the current value of the `trapflag` in the first MEX-file.
2. Calls the second MEX-file with the `trapflag` initialized to `0` within that file.
3. Restores the saved value of `trapflag` in the first MEX-file when the second MEX-file exits.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexsettrapflag.c`

See Also

`mexCallMATLAB`, `mexCallMATLABWithTrap`, `mexAtExit`, `mexErrMsgTxt`

Purpose

Allow clearing MEX-file from memory

C Syntax

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

Fortran Syntax

```
subroutine mexUnlock()
```

Description

By default, MEX-files are unlocked, meaning you can clear them at any time. Calling `mexLock` locks a MEX-file so that you cannot clear it from memory. Call `mexUnlock` to remove the lock.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexlock.c`
- `mexlockf.F`

See Also

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

mexWarnMsgIdAndTxt (C and Fortran)

Purpose

Display warning message with identifier

C Syntax

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

Fortran Syntax

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

Arguments

`warningid`
: String containing a MATLAB message identifier. For information on creating identifiers, see "Message Identifiers" in the MATLAB Programming Fundamentals documentation.

`warningmsg`
: String to display. In C, the string can include conversion specifications, used by the ANSI C `printf` function.

`...`
: In C, any arguments used in the message. Each argument must have a corresponding conversion specification.

Description

The `mexWarnMsgIdAndTxt` function writes a warning message to the MATLAB window. For more information, see the `warning` function syntax statement using a message identifier.

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

See Also

`mexErrMsgTxt`, `mexErrMsgIdAndTxt`, `mexWarnMsgTxt`

Purpose

Warning message

C Syntax

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

Fortran Syntax

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

Arguments

`warningmsg`
String containing the warning message to display

Description

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `yprime.c`
- `explore.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `fulltosparse.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`
- `mxsetnzmax.c`

See Also

`mexErrMsgTxt`, `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

mwIndex (C and Fortran)

Purpose

Type for index values

Description

`mwIndex` is a type that represents index values, such as indices into arrays. Use this function for cross-platform flexibility. By default, `mwIndex` is equivalent to `int` in C. When using the mex `-largeArrayDims` switch, `mwIndex` is equivalent to `size_t` in C. In Fortran, `mwIndex` is similarly equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

The C header file containing this type is:

```
#include "matrix.h"
```

In Fortran, `mwIndex` is a preprocessor macro. The Fortran header file containing this type is:

```
#include "fintrf.h"
```

See Also

`mex`, `mwSize`, `mwSignedIndex`

Purpose

Declare appropriate pointer type for platform

Description

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

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

The Fortran header file containing this type is:

```
#include "fintrf.h"
```

Examples

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

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

For additional examples, see the Fortran files with names ending in `.F` in the *`matlabroot`*`/extern/examples` folder.

mwSignedIndex (C and Fortran)

Purpose

Signed integer type for size values

Description

`mwSignedIndex` is a signed integer type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. By default, `mwSignedIndex` is equivalent to `ptrdiff_t` in C++. In Fortran, `mwSignedIndex` is similarly equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

The C header file containing this type is:

```
#include "matrix.h"
```

The Fortran header file containing this type is:

```
#include "fintrf.h"
```

See Also

`mwSize`

Purpose

Type for size values

Description

`mwSize` is a type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. By default, `mwSize` is equivalent to `int` in C. When using the mex `-largeArrayDims` switch, `mwSize` is equivalent to `size_t` in C. In Fortran, `mwSize` is similarly equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

The C header file containing this type is:

```
#include "matrix.h"
```

In Fortran, `mwSize` is a preprocessor macro. The Fortran header file containing this type is:

```
#include "fintrf.h"
```

See Also

`mex`, `mwIndex`, `mwSignedIndex`

mxAddField (C and Fortran)

Purpose

Add field to structure array

C Syntax

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

Fortran Syntax

```
integer*4 mxAddField(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

Arguments

`pm`
: Pointer to a structure `mxArray`

`fieldname`
: 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. Create the values with the `mxCreate*` functions and use `mxSetFieldByNumber` to set the individual values for the field.

See Also

`mxRemoveField`, `mxSetFieldByNumber`

Purpose

Type for a MATLAB array

Description

The fundamental type underlying MATLAB data. For information on how the MATLAB array works with MATLAB-supported variables, see "MATLAB Data" in the External Interfaces documentation.

`mxArray` is a C language opaque type.

All C and Fortran MEX-files start with a gateway routine, called `mexFunction`, which requires `mxArray` for both input and output parameters. For information about the C MEX-file gateway routine, see "C/C++ Source MEX-Files". For information about the Fortran version, see "Fortran Source MEX-Files".

Once you have MATLAB data in your MEX-file, use functions in the "MX Matrix Library" on page 1-2 to manipulate the data, and functions in the "MEX Library" on page 1-10 to perform operations in the MATLAB environment. You use `mxArray` to pass data to and from these functions.

Use any of the `mxCreate*` functions to create data, and the corresponding `mxDestroyArray` function to free memory.

The header file containing this type is:

```
#include "matrix.h"
```

Example

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatecharmatrixfromstr.c`

See Also

`mexFunction`, `mxClassID`, `mxCreateDoubleMatrix`, `mxCreateNumericArray`, `mxCreateString`, `mxDestroyArray`, `mxGetData`, `mxSetData`

mxArrayToString (C)

Purpose

Convert array to string

C Syntax

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

Arguments

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

Returns

C-style string. Returns NULL on failure. Possible reasons for failure include out of memory and specifying an `mxArray` that is not a string `mxArray`.

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 the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexatexit.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatecharmatrixfromstr.c`

- mxislogical.c

See Also

mxCreateCharArray, mxCreateCharMatrixFromStrings, mxCreateString, mxGetString

mxAssert (C)

Purpose

Check assertion value for debugging purposes

C Syntax

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

Arguments

`expr`
: Value of assertion

`error_message`
: Description of why assertion failed

Description

Like the ANSI C `assert` macro, `mxAssert` checks the value of an assertion, and continues execution only if the assertion holds. If `expr` evaluates to logical `1` (`true`), `mxAssert` does nothing. If `expr` evaluates to logical `0` (`false`), `mxAssert` prints an error to the MATLAB command window consisting of the failed assertion's expression, the filename and line number where the failed assertion occurred, and the `error_message` string. The `error_message` string allows you to specify a better description of why the assertion failed. Use an empty string if you do not want a description to follow the failed assertion message.

For information about MATLAB behavior after a failed assertion, see "Abnormal Termination" in the Desktop Tools and Development Environment documentation.

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

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.

See Also

mxAssertS

mxAssertS (C)

Purpose

Check assertion value without printing assertion text

C Syntax

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

Arguments

`expr`
Value of assertion

`error_message`
Description of why assertion failed

Description

`mxAssertS` is like `mxAssert`, except `mxAssertS` does not print the text of the failed assertion.

See Also

`mxAssert`

Purpose

Offset from first element to desired element

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to an `mxArray`

`nsubs`

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

subs

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

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

Returns

The number of elements, or *index*, between the start of the `mxArray` and the specified subscript. Many MX Matrix Library 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`, `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 first 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.

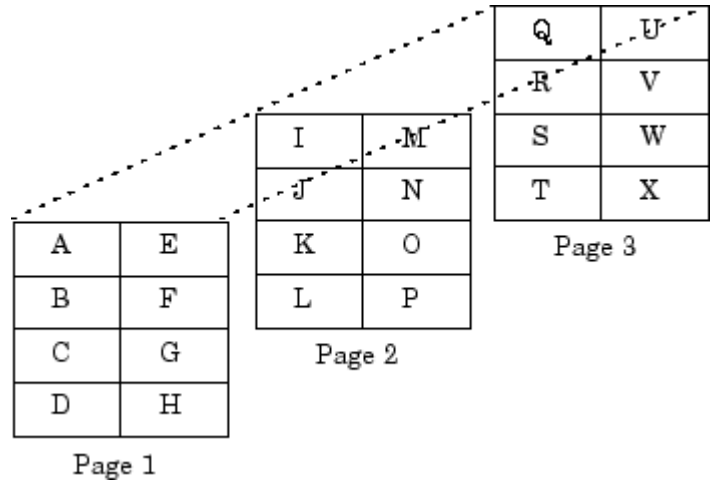
MATLAB uses a column-major numbering scheme to represent data elements internally. That means that MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on, through the last column. For example, suppose you create a `4-by-2` variable.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

A	E
B	F
C	G
D	H

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

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

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



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

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`

See Also

`mxGetCell`, `mxSetCell`

mxCalloc (C and Fortran)

Purpose

Allocate dynamic memory for array using MATLAB memory manager

C Syntax

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

Fortran Syntax

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

Arguments

`n`

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

`size`

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

Returns

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

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

Description

MATLAB applications must call `mxCalloc` rather than the ANSI C `calloc` function to allocate memory. In standalone applications, such as the MATLAB engine, `mxCalloc` calls the `calloc` function. 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 manager.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument

in `plhs[]` using the `mxSetPr` function, MATLAB is responsible for freeing the memory.

If you use the data internally, the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling this function. 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.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillSetData.c`
- `phonebook.c`
- `revord.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`
- `mxsetdimensions.c`

See Also

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

mxChar (C)

Purpose

Type for string `mxArray`

Description

A string `mxArray` stores its data elements as `mxChar` rather than as `char`.

The header file containing this type is:

```
#include "matrix.h"
```

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxmalloc.c`
- `mxcreatecharmatrixfromstr.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

`mxCreateCharArray`

Purpose

Enumerated value identifying class of `mxArray`

C Syntax

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

Constants

`mxUNKNOWN_CLASS`
: Undetermined class. You cannot specify this category for an `mxArray`; however, if `mxGetClassID` cannot identify the class, it returns this value.

`mxCELL_CLASS`
: Identifies a cell `mxArray`.

`mxSTRUCT_CLASS`
: Identifies a structure `mxArray`.

`mxLOGICAL_CLASS`
: Identifies a logical `mxArray`, an `mxArray` of `mxLogical` data.

`mxCHAR_CLASS`
: Identifies a string `mxArray`, an `mxArray` whose data is represented as `mxChar`.

`mxVOID_CLASS`
: Reserved.

`mxDOUBLE_CLASS`
: Identifies a numeric `mxArray` whose data is stored as double-precision, floating-point numbers.

`mxSINGLE_CLASS`
: Identifies a numeric `mxArray` whose data is stored as single-precision, floating-point numbers.

`mxINT8_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 8-bit integers.

`mxUINT8_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 8-bit integers.

`mxINT16_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 16-bit integers.

`mxUINT16_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 16-bit integers.

`mxINT32_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 32-bit integers.

`mxUINT32_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 32-bit integers.

`mxINT64_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 64-bit integers.

`mxUINT64_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 64-bit integers.

`mxFUNCTION_CLASS`
Identifies a function handle `mxArray`.

Description

Various MX Matrix Library functions require or return an `mxClassID` argument. `mxClassID` identifies the way in which the `mxArray` represents its data elements.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

`mxGetClassID` , `mxCreateNumericArray`

mxClassIDFromClassName (Fortran)

Purpose

Identifier corresponding to class

Fortran Syntax

```
integer*4 mxClassIDFromClassName(classname)
character*(*) classname
```

Arguments

`classname`

`character` array specifying a MATLAB class name. Use one of the strings from the following table.

Returns

Numeric identifier used internally by MATLAB software to represent the MATLAB class, `classname`. Returns `unknown` if `classname` is not a recognized MATLAB class.

Description

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

For a list of valid `classname` choices, see the `mxIsClass` reference page.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `matsqint8.F`

See Also

`mxGetClassName`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxIsClass`

Purpose

Flag specifying whether mxArray has imaginary components

C Syntax

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

Constants

`mxREAL`
: Identifies an `mxArray` with no imaginary components.

`mxCOMPLEX`
: Identifies an `mxArray` with imaginary components.

Description

Various MX Matrix Library functions require an `mxComplexity` argument. You can set an `mxComplex` argument to either `mxREAL` or `mxCOMPLEX`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`

See Also

`mxCreateNumericArray`, `mxCreateDoubleMatrix`, `mxCreateSparse`

mxCopyCharacterToPtr (Fortran)

Purpose

Copy `character` values from Fortran array to pointer array

Fortran Syntax

```
subroutine mxCopyCharacterToPtr(y, px, n)
character*(*) y
mwPointer px
mwSize n
```

Arguments

`y`
: `character` Fortran array

`px`
: Pointer to `character` or `name` array

`n`
: Number of elements to copy

Description

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

See Also

`mxCopyPtrToCharacter`, `mxCreateCharArray`, `mxCreateString`, `mxCreateCharMatrixFromStrings`

Purpose

Copy COMPLEX*16 values from Fortran array to pointer array

Fortran Syntax

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

Arguments

`y`
: COMPLEX*16 Fortran array

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

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

`n`
: Number of elements to copy

Description

`mxCopyComplex16ToPtr` copies `n` `COMPLEX*16` values from the Fortran `COMPLEX*16` array `y` into the MATLAB arrays pointed to by `pr` and `pi`.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `convec.F`

See Also

`mxCopyPtrToComplex16`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

mxCopyComplex8ToPtr (Fortran)

Purpose

Copy `COMPLEX*8` values from Fortran array to pointer array

Fortran Syntax

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

Arguments

`y`
: `COMPLEX*8` Fortran array

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

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

`n`
: Number of elements to copy

Description

`mxCopyComplex8ToPtr` copies `n` `COMPLEX*8` values from the Fortran `COMPLEX*8` array `y` into the MATLAB arrays pointed to by `pr` and `pi`.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

See Also

`mxCopyPtrToComplex8`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose

Copy INTEGER*1 values from Fortran array to pointer array

Fortran Syntax

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

Arguments

`y`
: INTEGER*1 Fortran array

`px`
: Pointer to the real or imaginary data of the array

`n`
: Number of elements to copy

Description

`mxCopyInteger1ToPtr` copies `n` INTEGER*1 values from the Fortran INTEGER*1 array `y` into the MATLAB array pointed to by `px`, either a real or an imaginary array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `matsqint8.F`

See Also

`mxCopyPtrToInteger1`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

mxCopyInteger2ToPtr (Fortran)

Purpose

Copy INTEGER*2 values from Fortran array to pointer array

Fortran Syntax

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

Arguments

`y`
: INTEGER*2 Fortran array

`px`
: Pointer to the real or imaginary data of the array

`n`
: Number of elements to copy

Description

`mxCopyInteger2ToPtr` copies `n` INTEGER*2 values from the Fortran INTEGER*2 array `y` into the MATLAB array pointed to by `px`, either a real or an imaginary array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

See Also

`mxCopyPtrToInteger2`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Purpose

Copy INTEGER*4 values from Fortran array to pointer array

Fortran Syntax

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

Arguments

`y`
: INTEGER*4 Fortran array

`px`
: Pointer to the real or imaginary data of the array

`n`
: Number of elements to copy

Description

`mxCopyInteger4ToPtr` copies `n` INTEGER*4 values from the Fortran INTEGER*4 array `y` into the MATLAB array pointed to by `px`, either a real or an imaginary array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

See Also

`mxCopyPtrToInteger4`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

mxCopyPtrToCharacter (Fortran)

Purpose

Copy `character` values from pointer array to Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToCharacter(px, y, n)
mwPointer px
character*(*) y
mwSize n
```

Arguments

`px`
: Pointer to `character` or `name` array

`y`
: `character` Fortran array

`n`
: Number of elements to copy

Description

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matdemo2.F`

See Also

`mxCopyCharacterToPtr`, `mxCreateCharArray`, `mxCreateString`, `mxCreateCharMatrixFromStrings`

Purpose

Copy COMPLEX*16 values from pointer array to Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToComplex16(pr, pi, y, n)
mwPointer pr, pi
complex*16 y(n)
mwSize n
```

Arguments

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

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

`y`
: COMPLEX*16 Fortran array

`n`
: Number of elements to copy

Description

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

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- convec.F

See Also

mxCopyComplex16ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyPtrToComplex8 (Fortran)

Purpose

Copy `COMPLEX*8` values from pointer array to Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToComplex8(pr, pi, y, n)
mwPointer pr, pi
complex*8 y(n)
mwSize n
```

Arguments

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

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

`y`
: `COMPLEX*8` Fortran array

`n`
: Number of elements to copy

Description

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

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

See Also

`mxCopyComplex8ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose

Copy INTEGER*1 values from pointer array to Fortran array

Fortran Syntax

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

Arguments

px
: Pointer to the real or imaginary data of the array

y
: INTEGER*1 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToInteger1 copies n INTEGER*1 values from the MATLAB array pointed to by px, either a real or imaginary array, into the Fortran INTEGER*1 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- matsqint8.F

See Also

mxCopyInteger1ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

mxCopyPtrToInteger2 (Fortran)

Purpose

Copy INTEGER*2 values from pointer array to Fortran array

Fortran Syntax

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

Arguments

px
: Pointer to the real or imaginary data of the array

y
: INTEGER*2 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToInteger2 copies n INTEGER*2 values from the MATLAB array pointed to by px, either a real or an imaginary array, into the Fortran INTEGER*2 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

See Also

mxCopyInteger2ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

Purpose

Copy INTEGER*4 values from pointer array to Fortran array

Fortran Syntax

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

Arguments

px
: Pointer to the real or imaginary data of the 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 a real or an imaginary array, into the Fortran INTEGER*4 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

See Also

mxCopyInteger4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

mxCopyPtrToPtrArray (Fortran)

Purpose

Copy pointer values from pointer array to Fortran array

Fortran Syntax

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

Arguments

`px`
: Pointer to pointer array

`y`
: Fortran array of `mwPointer` values

`n`
: Number of pointers to copy

Description

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matdemo2.F`

See Also

`matGetDir`, `mxCopyPtrToCharacter`

Purpose

Copy REAL*4 values from pointer array to Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToReal4(px, y, n)
mwPointer px
real*4 y(n)
mwSize n
```

Arguments

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

y
: REAL*4 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToReal4 copies n REAL*4 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*4 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

See Also

mxCopyReal4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyPtrToReal8 (Fortran)

Purpose

Copy REAL*8 values from pointer array to Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToReal8(px, y, n)
mwPointer px
real*8 y(n)
mwSize n
```

Arguments

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

y
: REAL*8 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToReal8 copies n REAL*8 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*8 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- fengdemo.F

See the following examples in *matlabroot*/extern/examples/refbook.

- timestwo.F
- xtimesy.F

See Also

mxCopyReal8ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

Purpose

Copy REAL*4 values from Fortran array to pointer array

Fortran Syntax

```
subroutine mxCopyReal4ToPtr(y, px, n)
real*4 y(n)
mwPointer px
mwSize n
```

Arguments

y
: REAL*4 Fortran array

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

n
: Number of elements to copy

Description

mxCopyReal4ToPtr copies n REAL*4 values from the Fortran REAL*4 array y into the MATLAB array pointed to by px, either a pr or pi array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

See Also

mxCopyPtrToReal4, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyReal8ToPtr (Fortran)

Purpose

Copy REAL*8 values from Fortran array to pointer array

Fortran Syntax

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

Arguments

y
: REAL*8 Fortran array

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

n
: Number of elements to copy

Description

mxCopyReal8ToPtr 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.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

Examples

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matdemo1.F
- fengdemo.F

See the following examples in *matlabroot*/extern/examples/refbook.

- timestwo.F
- xtimesy.F

See Also

mxCopyPtrToReal8, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

Purpose

Create unpopulated N-D cell mxArray

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateCellArray(ndim, dims)
mwSize ndim, dims
```

Arguments

ndim

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

dims

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

Returns

Pointer to the created mxArray, if successful. If unsuccessful in a standalone (non-MEX-file) application, returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Use mxCreateCellArray to create a cell mxArray with size defined by ndim and dims. For example, in C, 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;
```

In Fortran, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set:

```
ndim = 3;
```

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

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

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array has the dimensions 4-by-1-by-7.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See Also

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

Purpose

Create unpopulated 2-D cell `mxArray`

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateCellMatrix(m, n)
mwSize m, n
```

Arguments

`m`
: Number of rows

`n`
: Number of columns

Returns

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

Description

Use `mxCreateCellMatrix` to create an `m`-by-`n` two-dimensional cell `mxArray`. The created cell `mxArray` is unpopulated; `mxCreateCellMatrix` initializes each cell to `NULL` in C (`0` in Fortran). To put data into cells, call `mxSetCell`.

`mxCreateCellMatrix` is identical to `mxCreateCellArray` except that `mxCreateCellMatrix` can create two-dimensional `mxArrays` only, but `mxCreateCellArray` can create `mxArrays` having any number of dimensions greater than 1.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`

See Also

`mxCreateCellArray`

mxCreateCharArray (C and Fortran)

Purpose

Create unpopulated N-D string `mxArray`

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateCharArray(ndim, dims)
mwSize ndim, dims
```

Arguments

`ndim`

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`

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

Returns

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

Description

Call `mxCreateCharArray` to create an N-dimensional string `mxArray`. The created `mxArray` is unpopulated; that is, `mxCreateCharArray` initializes each cell to `NULL` in C (`0` in Fortran).

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array has the dimensions `4`-by-`1`-by-`7`.

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatecharmatrixfromstr.c

See Also

mxCreateCharMatrixFromStrings, mxCreateString

mxCreateCharMatrixFromStrings (C and Fortran)

Purpose

Create populated 2-D string mxArray

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateCharMatrixFromStrings(m, str)
mwSize m
character*(*) str(m)
```

Arguments

`m`

Number of rows in the created string mxArray. The value you specify for m is the number of strings in str.

`str`

In C, an array of strings containing at least m strings. In Fortran, a character*n array of size m, where each element of the array is n bytes.

Returns

Pointer to the created mxArray, if successful. If unsuccessful in a standalone (non-MEX-file) application, returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the mxArray. 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. In C, the created mxArray has dimensions m-by-max, where max is the length of the longest string in str. In Fortran, the created mxArray has dimensions m-by-n, where n is the number of characters in str(i).

String mxArrays represent their data elements as mxChar rather than as C char.

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatecharmatrixfromstr.c

See Also mxCreateCharArray, mxCreateString, mxGetString

mxCreateDoubleMatrix (C and Fortran)

Purpose

Create 2-D, double-precision, floating-point mxArray initialized to 0

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateDoubleMatrix(m, n, ComplexFlag)
mwSize m, n
integer*4 ComplexFlag
```

Arguments

m
: Number of rows

n
: Number of columns

ComplexFlag
: If the mxArray you are creating is to contain imaginary data, set ComplexFlag to mxCOMPLEX in C (1 in Fortran). Otherwise, set ComplexFlag to mxREAL in C (0 in Fortran).

Returns

Pointer to the created mxArray, if successful. If unsuccessful in a standalone (non-MEX-file) application, returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and returns control to the MATLAB prompt. The function 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 in C (1 in Fortran), mxCreateDoubleMatrix also initializes each element in the pi array to 0.

If you set ComplexFlag to mxREAL in C (0 in Fortran), mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements. If you set ComplexFlag to mxCOMPLEX in C (1 in Fortran), 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 the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `convec.c`
- `findnz.c`
- `matrixDivide.c`
- `sincall.c`
- `timestwo.c`
- `timestwoalt.c`
- `xtimesy.c`

For Fortran examples, see:

- `convec.F`
- `dblmat.F`
- `matsq.F`
- `timestwo.F`
- `xtimesy.F`

See Also

`mxCreateNumericArray`

mxCreateDoubleScalar (C and Fortran)

Purpose

Create scalar, double-precision array initialized to specified value

C Syntax

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

Fortran Syntax

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

Arguments

`value`
Value to which you want to initialize the array

Returns

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

Description

Call `mxCreateDoubleScalar` to create a scalar double `mxArray`. When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

Alternatives

C Language

In C, you can replace the statements:

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

with a call to `mxCreateDoubleScalar`:

```
pa = mxCreateDoubleScalar(value);
```

Fortran Language

In Fortran, you can replace the statements:

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

with a call to `mxCreateDoubleScalar`:

```
pm = mxCreateDoubleScalar(value)
```

See Also

`mxGetPr`, `mxCreateDoubleMatrix`

mxCreateLogicalArray (C)

Purpose

Create N-D logical mxArray initialized to false

C Syntax

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

Arguments

ndim

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

dims

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxArray. There are ndim elements in the dims array.

Returns

Pointer to the created mxArray, if successful. If unsuccessful in a standalone (non-MEX-file) application, returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Call mxCreateLogicalArray to create an N-dimensional mxArray of mxLogical elements. After creating the mxArray, mxCreateLogicalArray initializes all its elements to logical 0. mxCreateLogicalArray differs from mxCreateLogicalMatrix in that the latter can create two-dimensional arrays only.

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

MATLAB automatically removes any trailing singleton dimensions specified in the dims argument. For example, if ndim equals 5 and dims equals [4 1 7 1 1], the resulting array has the dimensions 4-by-1-by-7.

See Also

mxCreateLogicalMatrix, mxCreateSparseLogicalMatrix, mxCreateLogicalScalar

mxCreateLogicalMatrix (C)

Purpose

Create 2-D, logical `mxArray` initialized to `false`

C Syntax

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

Arguments

`m`
: Number of rows

`n`
: Number of columns

Returns

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

Description

Use `mxCreateLogicalMatrix` to create an `m`-by-`n` `mxArray` of `mxLogical` elements. `mxCreateLogicalMatrix` initializes each element in the array to logical `0`.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxislogical.c`

See Also

`mxCreateLogicalArray`, `mxCreateSparseLogicalMatrix`, `mxCreateLogicalScalar`

Purpose

Create scalar, logical `mxArray`

C Syntax

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

Arguments

`value`
Logical value to which you want to initialize the array

Returns

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

Description

Call `mxCreateLogicalScalar` to create a scalar logical `mxArray`. `mxCreateLogicalScalar` is a convenience function that replaces the following code:

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

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

See Also

`mxCreateLogicalArray`, `mxCreateLogicalMatrix`, `mxIsLogicalScalar`, `mxIsLogicalScalarTrue`, `mxGetLogicals`, `mxDestroyArray`

mxCreateNumericArray (C and Fortran)

Purpose

Create unpopulated N-D numeric mxArray

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateNumericArray(ndim, dims, classid,
  ComplexFlag)
mwSize ndim, dims
integer*4 classid, ComplexFlag
```

Arguments

`ndim`

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

`dims`

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

`classid`

Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` in C causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer. In Fortran, use the function `mxClassIDFromClassName` to derive the `classid` value from a MATLAB class name. See the Description section for more information.

`ComplexFlag`

If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran). Otherwise, set `ComplexFlag` to `mxREAL` in C (`0` in Fortran).

Returns

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

Description

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

- All data elements in `mxCreateDoubleMatrix` are double-precision, floating-point numbers. The data elements in `mxCreateNumericArray` can 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.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array has the dimensions 4-by-1-by-7.

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

MATLAB Class Name	C `classid` Value	Fortran Type
`int8`	`mxINT8_CLASS`	`BYTE`
`uint8`	`mxUINT8_CLASS`	

MATLAB Class Name	C `classid` Value	Fortran Type
int16	mxINT16_CLASS	INTEGER*2
uint16	mxUINT16_CLASS	
int32	mxINT32_CLASS	INTEGER*4
uint32	mxUINT32_CLASS	
int64	mxINT64_CLASS	INTEGER*8
uint64	mxUINT64_CLASS	
single	mxSINGLE_CLASS	REAL*4
double	mxDOUBLE_CLASS	REAL*8
single, with imaginary components	mxSINGLE_CLASS	COMPLEX*8
double, with imaginary components	mxDOUBLE_CLASS	COMPLEX*16

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- phonebook.c
- doubleelement.c
- matrixDivide.c
- matsqint8.F

See the following examples in *matlabroot*/extern/examples/mx.

- mxisfinite.c

See Also

mxClassId, mxClassIdFromClassName, mxComplexity, mxCreateNumericMatrix

Purpose

Create numeric matrix initialized to 0

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateNumericMatrix(m, n, classid,
  ComplexFlag)
mwSize m, n
integer*4 classid, ComplexFlag
```

Arguments

`m`
: Number of rows

`n`
: Number of columns

`classid`
: Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` in C causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer. In Fortran, use the function `mxClassIDFromClassName` to derive the `classid` value from a MATLAB class name.

`ComplexFlag`
: If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran). Otherwise, set `ComplexFlag` to `mxREAL` in C (`0` in Fortran).

Returns

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

Description

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

creating the mxArray, mxCreateNumericMatrix initializes all its real data elements to 0. If ComplexFlag equals mxCOMPLEX in C (1 in Fortran), mxCreateNumericMatrix also initializes all its imaginary data elements to 0. mxCreateNumericMatrix allocates dynamic memory to store the created mxArray. When you finish using the mxArray, call mxDestroyArray to destroy it.

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

MATLAB Class Name	C classid Value	Fortran Type
int8	mxINT8_CLASS	BYTE
uint8	mxUINT8_CLASS	
int16	mxINT16_CLASS	INTEGER*2
uint16	mxUINT16_CLASS	
int32	mxINT32_CLASS	INTEGER*4
uint32	mxUINT32_CLASS	
int64	mxINT64_CLASS	INTEGER*8
uint64	mxUINT64_CLASS	
single	mxSINGLE_CLASS	REAL*4
double	mxDOUBLE_CLASS	REAL*8
single, with imaginary components	mxSINGLE_CLASS	COMPLEX*8
double, with imaginary components	mxDOUBLE_CLASS	COMPLEX*16

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- arrayFillGetPr.c

The following Fortran statements create a 4-by-3 matrix of REAL*4 elements having no imaginary components:

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

See Also

mxClassId, mxClassIdFromClassName, mxComplexity, mxCreateNumericArray

mxCreateSparse (C and Fortran)

Purpose

Create 2-D unpopulated sparse mxArray

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateSparse(m, n, nzmax, ComplexFlag)
mwSize m, n, nzmax
integer*4 ComplexFlag
```

Arguments

m
: Number of rows

n
: Number of columns

nzmax
: Number of elements that mxCreateSparse should allocate to hold the pr, ir, and, if ComplexFlag is mxCOMPLEX in C (1 in Fortran), pi arrays. Set the value of nzmax to be greater than or equal to the number of nonzero elements you plan to put into the mxArray, but make sure that nzmax is less than or equal to m*n.

ComplexFlag
: If the mxArray you are creating is to contain imaginary data, set ComplexFlag to mxCOMPLEX in C (1 in Fortran). Otherwise, set ComplexFlag to mxREAL in C (0 in Fortran).

Returns

Pointer to the created sparse double mxArray if successful, and NULL in C (0 in Fortran) otherwise. The most likely reason for failure is insufficient free heap space. If that happens, try reducing nzmax, m, or n.

Description

Call mxCreateSparse to create an unpopulated sparse double mxArray. The returned sparse mxArray contains no sparse information and cannot be passed as an argument to any MATLAB sparse functions. To make the returned sparse mxArray useful, you must initialize the pr, ir, jc, and (if it exists) pi arrays.

mxCreateSparse allocates space for:

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

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

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- fulltosparse.c
- fulltosparse.F

See Also

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

mxCreateSparseLogicalMatrix (C)

Purpose

Create unpopulated 2-D, sparse, logical `mxArray`

C Syntax

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

Arguments

`m`
: Number of rows

`n`
: Number of columns

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

Returns

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

Description

Use `mxCreateSparseLogicalMatrix` to create an `m`-by-`n` `mxArray` of `mxLogical` elements. `mxCreateSparseLogicalMatrix` initializes each element in the array to logical `0`.

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

See Also

`mxCreateLogicalArray`, `mxCreateLogicalMatrix`, `mxCreateLogicalScalar`, `mxCreateSparse`, `mxIsLogical`

Purpose

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

C Syntax

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

Fortran Syntax

```
mwPointer mxCreateString(str)
character*(*) str
```

Arguments

`str`
String that is to serve as the mxArray's initial data

Returns

Pointer to the created string mxArray if successful, and NULL in C (0 in Fortran) otherwise. The most likely cause of failure is insufficient free heap space.

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 the following examples in *matlabroot*/extern/examples/refbook.

- revord.c
- revord.F

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatestructarray.c
- mxisclass.c

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matdemo1.F

See Also `mxCreateCharMatrixFromStrings`, `mxCreateCharArray`

Purpose

Create unpopulated N-D structure mxArray

C Syntax

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

Fortran Syntax

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

Arguments

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

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

`nfields`
: Number of fields in each element

`fieldnames`
: List of field names

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

Returns

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

Description

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

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

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatestructarray.c`

See Also

`mxDestroyArray`, `mxAddField`, `mxRemoveField`, `mxSetField`, `mxSetFieldByNumber`

Purpose

Create unpopulated 2-D structure `mxArray`

C Syntax

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

Fortran Syntax

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

Arguments

`m`
: Number of rows. This must be a positive integer.

`n`
: Number of columns. This must be a positive integer.

`nfields`
: Number of fields in each element.

`fieldnames`
: List of field names.

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

Returns

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

Description

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

C Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- phonebook.c

See Also

mxCreateStructArray

Purpose

Free dynamic memory allocated by mxCreate* functions

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to the mxArray you want to free

Description

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

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- matrixDivide.c
- matrixDivideComplex.c
- sincall.c
- sincall.F

See the following examples in *matlabroot*/extern/examples/mex.

- mexcallmatlab.c
- mexgetarray.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxisclass.c

- mxcreatecellmatrixf.F

See Also

mxCalloc, mxMalloc, mxFree, mexMakeArrayPersistent, mexMakeMemoryPersistent

Purpose

Make deep copy of array

C Syntax

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

Fortran Syntax

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

Arguments

`in`

Pointer to the `mxArray` you want to copy

Returns

Pointer to a copy of the array.

Description

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexget.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`
- `mxgetinf.c`
- `mxsetdimensions.c`
- `mxsetdimensionsf.F`

- mxsetnzmax.c

Purpose

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

C Syntax

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

Fortran Syntax

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

Arguments

`ptr`

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

Description

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

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

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

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

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

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`
- `mxcreatecharmatrixfromstr.c`
- `mxisfinite.c`
- `mxmalloc.c`
- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillGetPrDynamicData.c`
- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

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

Purpose

Contents of mxArray cell

C Syntax

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

Fortran Syntax

```
mwPointer mxGetCell(pm, index)
mwPointer pm
mwIndex index
```

Arguments

`pm`

Pointer to a cell mxArray

`index`

Number of elements in the cell mxArray between the first element and the desired one. See mxCalcSingleSubscript for details on calculating an index in a multidimensional cell array.

Returns

Pointer to the ith cell mxArray if successful, and NULL in C (0 in Fortran) otherwise. Causes of failure include

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

Description

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

Note Inputs to a MEX-file are constant read-only mxArrays. Do not modify the inputs. Using mxSetCell* or mxSetField* functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

Examples

See the following examples in *matlabroot*/extern/examples/mex.

- explore.c

See Also

mxCreateCellArray, mxIsCell, mxSetCell

Purpose

Pointer to character array data

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

Pointer to the first character in the `mxArray`. Returns `NULL` if the specified array is not a character array.

Description

Call `mxGetChars` to access the first character in the `mxArray` that `array_ptr` points to. Once you have the starting address, you can access any other element in the `mxArray`.

See Also

`mxGetString`

mxGetClassID (C and Fortran)

Purpose

Class of `mxArray`

C Syntax

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

Fortran Syntax

```
integer*4 mxGetClassID(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Numeric identifier of the class (category) of the `mxArray` that `pm` points to. For a list of C-language class identifiers, see the `mxClassID` reference page.

Description

Use `mxGetClassId` to determine the class of an `mxArray`. The class of an `mxArray` identifies the kind of data the `mxArray` is holding. For example, if `pm` points to a logical `mxArray`, then `mxGetClassId` returns `mxLOGICAL_CLASS` (in C).

`mxGetClassId` is like `mxGetClassName`, except that the former returns the class as an integer identifier and the latter returns the class as a string.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See Also

`mxClassID`, `mxGetClassName`

Purpose

Class of mxArray as string

C Syntax

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

Fortran Syntax

```
character*(*) mxGetClassName(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Class (as a string) of the mxArray pointed to by pm.

Description

Call mxGetClassName to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For example, if pm points to a logical mxArray, mxGetClassName returns logical.

mxGetClassID is like mxGetClassName, except that the former returns the class as an integer identifier, as listed in the mxClassID reference page, and the latter returns the class as a string, as listed in the mxIsClass reference page.

Examples

See the following examples in *matlabroot*/extern/examples/mex.

- mexfunction.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxisclass.c

See Also

mxGetClassID, mxIsClass

mxGetData (C and Fortran)

Purpose

Pointer to real data

C Syntax

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

Fortran Syntax

```
mwPointer mxGetData(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`

Returns

Pointer to the first element of the real data. Returns `NULL` in C (`0` in Fortran) if there is no real data.

Description

Like `mxGetPr`, except that in C, `mxGetData` returns a `void *`.

To copy values from the returned pointer to Fortran, use one of the `mxCopyPtrTo*` functions in the following manner:

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `matrixDivide.c`
- `matrixDivideComplex.c`
- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatecharmatrixfromstr.c`
- `mxisfinite.c`

See Also

`mxGetImagData`, `mxGetPr`

Purpose

Pointer to dimensions array

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to an `mxArray`.

Returns

Pointer to the first element in the dimensions array. Each integer in the dimensions array represents the number of elements in a particular dimension. The array is not `NULL` terminated.

Description

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

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxisfinite.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `findnz.c`

- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

`mxGetNumberOfDimensions`

Purpose

Number of bytes required to store each data element

C Syntax

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

Fortran Syntax

```
mwPointer mxGetElementSize(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

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

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

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `doubleelement.c`

- phonebook.c

See Also

mxGetM, mxGetN

Purpose

Value of `eps`

C Syntax

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

Fortran Syntax

```
real*8 mxGetEps
```

Returns

Value of the MATLAB `eps` variable

Description

Call `mxGetEps` to return the value of the MATLAB `eps` variable. This variable holds the distance from 1.0 to the next largest floating-point number. As such, it is a measure of floating-point accuracy. The MATLAB `pinv` and `rank` functions use `eps` as a default tolerance.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxgeteps.c`
- `mxgetepsf.F`

See Also

`mxGetInf`, `mxGetNan`

mxGetField (C and Fortran)

Purpose

Field value, given field name and index, into structure array

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to a structure `mxArray`

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`fieldname`

Name of the field whose value you want to extract.

Returns

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

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

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

Description

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

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

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

Note Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

In C, calling:

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

is equivalent to calling:

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

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

In Fortran, calling:

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

is equivalent to calling:

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

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

See Also

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

Purpose

Field value, given field number and index, into structure array

C Syntax

```
#include "matrix.h"
mxArray *mxGetFieldByNumber(const mxArray *pm, mwIndex index,
         int fieldnumber);
```

Fortran Syntax

```
mwPointer mxGetFieldByNumber(pm, index, fieldnumber)
mwPointer pm
mwIndex index
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for more details on calculating an `index`.

`fieldnumber`

Position of the field whose value you want to extract

In C, the first field within each element has a field number of `0`, the second field has a field number of `1`, and so on. The last field has a field number of `N-1`, where `N` is the number of fields.

In Fortran, the first field within each element has a field number of `1`, the second field has a field number of `2`, and so on. The last field has a field number of `N`, where `N` is the number of fields.

mxGetFieldByNumber (C and Fortran)

Returns

Pointer to the `mxArray` in the specified field for the desired element, on success. Returns `NULL` in C (`0` in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include:

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

Description

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

Note Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

In C, calling:

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

is equivalent to calling:

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

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

In Fortran, calling:

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

is equivalent to calling:

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisclass.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

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

mxGetFieldNameByNumber (C and Fortran)

Purpose

Field name, given field number, in structure array

C Syntax

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

Fortran Syntax

```
character*(*) mxGetFieldNameByNumber(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

Position of the desired field. For instance, in C, to get the name of the first field, set `fieldnumber` to 0; to get the name of the second field, set `fieldnumber` to 1; and so on. In Fortran, to get the name of the first field, set `fieldnumber` to `1`; to get the name of the second field, set `fieldnumber` to `2`; and so on.

Returns

Pointer to the `n`th field name, on success. Returns `NULL` in C (`0` in Fortran) on failure. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.
- Specifying a value of `fieldnumber` outside the bounds of the number of fields in the structure `mxArray`. In C, `fieldnumber` 0 represents the first field, and `fieldnumber` `N-1` represents the last field, where `N` is the number of fields in the structure `mxArray`. In Fortran, `fieldnumber` `1` represents the first field, and `fieldnumber` `N` represents the last field.

Description

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

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

Consider a MATLAB structure initialized to:

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

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisclass.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

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

mxGetFieldNumber (C and Fortran)

Purpose

Field number, given field name, in structure array

C Syntax

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

Fortran Syntax

```
integer*4 mxGetFieldNumber(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

Arguments

`pm`
Pointer to a structure `mxArray`

`fieldname`
Name of a field in the structure `mxArray`

Returns

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

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

Description

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

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

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

Calling:

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

is equivalent to calling:

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatestructarray.c`

See Also

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

mxGetImagData (C and Fortran)

Purpose

Pointer to imaginary data of `mxArray`

C Syntax

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

Fortran Syntax

```
mwPointer mxGetImagData(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Pointer to the first element of the imaginary data. Returns `NULL` in C (`0` in Fortran) if there is no imaginary data or if there is an error.

Description

This function is like `mxGetPi`, except that in C it returns a `void *`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See Also

`mxGetData`, `mxGetPi`

Purpose

Value of infinity

C Syntax

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

Fortran Syntax

```
real*8 mxGetInf
```

Returns

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. Your system specifies the value of `inf`; you cannot modify it.

Operations that return infinity include:

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxgetinf.c`

See Also

`mxGetEps`, `mxGetNaN`

mxGetIr (C and Fortran)

Purpose

ir array of sparse matrix

C Syntax

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

Fortran Syntax

```
mwPointer mxGetIr(pm)
mwPointer pm
```

Arguments

pm

Pointer to a sparse mxArray

Returns

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

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

Description

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

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

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

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- fulltosparse.c
- fulltosparse.F

See the following examples in *matlabroot*/extern/examples/mx.

- `mxsetdimensions.c`
- `mxsetnzmax.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

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

mxGetJc (C and Fortran)

Purpose

`jc` array of sparse matrix

C Syntax

```
#include "matrix.h"
mwIndex *mxGetJc(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetJc(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a sparse `mxArray`

Returns

Pointer to the first element in the `jc` array, if successful, and `NULL` in C (`0` in Fortran) otherwise. Possible causes of failure include

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

Description

Use `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 the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `fulltosparse.c`
- `fulltosparse.F`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxgetnzmax.c`
- `mxsetdimensions.c`
- `mxsetnzmax.c`

See the following examples in *matlabroot*/extern/examples/mex.

- explore.c

See Also

mxGetIr, mxGetNzmax, mxSetIr, mxSetJc, mxSetNzmax

mxGetLogicals (C)

Purpose

Pointer to logical array data

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

Pointer to the first logical element in the `mxArray`. The result is unspecified if the `mxArray` is not a logical array.

Description

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxislogical.c`

See Also

`mxCreateLogicalArray`, `mxCreateLogicalMatrix`, `mxCreateLogicalScalar`, `mxIsLogical`, `mxIsLogicalScalar`, `mxIsLogicalScalarTrue`

Purpose

Number of rows in mxArray

C Syntax

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

Fortran Syntax

```
mwPointer mxGetM(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Number of rows in the mxArray to which pm 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 pm points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, mxGetM returns 8.

Note Fortran does not have an equivalent of size_t. mwPointer is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- convec.c
- fulltosparse.c
- matrixDivide.c
- matrixDivideComplex.c
- revord.c
- timestwo.c

- `xtimesy.c`

For Fortran examples, see:

- `convec.F`
- `dblmat.F`
- `fulltosparse.F`
- `matsq.F`
- `timestwo.F`
- `xtimesy.F`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxmalloc.c`
- `mxsetdimensions.c`
- `mxgetnzmax.c`
- `mxsetnzmax.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`
- `mexget.c`
- `mexlock.c`
- `mexsettrapflag.c`
- `yprime.c`

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matdemo2.F`

See Also

`mxGetN`, `mxSetM`, `mxSetN`

Purpose

Number of columns in `mxArray`

C Syntax

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

Fortran Syntax

```
mwPointer mxGetN(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`

Returns

Number of columns in the `mxArray`.

Description

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

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

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

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `convec.c`
- `fulltosparse.c`

- revord.c
- timestwo.c
- xtimesy.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxmalloc.c
- mxsetdimensions.c
- mxgetnzmax.c
- mxsetnzmax.c

See the following examples in *matlabroot*/extern/examples/mex.

- explore.c
- mexget.c
- mexlock.c
- mexsettrapflag.c
- yprime.c

See the following examples in *matlabroot*/extern/examples/eng_mat.

- matdemo2.F

See Also

mxGetM, mxGetDimensions, mxSetM, mxSetN

Purpose

Value of NaN (Not-a-Number)

C Syntax

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

Fortran Syntax

```
real*8 mxGetNaN
```

Returns

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

Your system specifies the value of Not-a-Number. You cannot modify it.

C Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxgetinf.c

See Also

mxGetEps, mxGetInf

mxGetNumberOfDimensions (C and Fortran)

Purpose

Number of dimensions in `mxArray`

C Syntax

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

Fortran Syntax

```
mwSize mxGetNumberOfDimensions(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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 the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `findnz.c`
- `fulltosparse.c`
- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxisfinite.c`

See Also mxSetM, mxSetN, mxGetDimensions

mxGetNumberOfElements (C and Fortran)

Purpose

Number of elements in `mxArray`

C Syntax

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

Fortran Syntax

```
mwPointer mxGetNumberOfElements(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Number of elements in the specified `mxArray`

Description

`mxGetNumberOfElements` tells you how many elements an array has. For example, if the dimensions of an array are 3-by-5-by-10, `mxGetNumberOfElements` returns the number `150`.

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `findnz.c`
- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxgetepsf.F`
- `mxgetinf.c`

- `mxisfinite.c`
- `mxsetdimensions.c`
- `mxsetdimensionsf.F`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

`mxGetDimensions`, `mxGetM`, `mxGetN`, `mxGetClassID`, `mxGetClassName`

mxGetNumberOfFields (C and Fortran)

Purpose

Number of fields in structure `mxArray`

C Syntax

```
#include "matrix.h"
int mxGetNumberOfFields(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxGetNumberOfFields(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a structure `mxArray`

Returns

Number of fields, on success. Returns `0` on failure. The most common cause of failure is that `pm` is not a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` is a structure.

Description

Call `mxGetNumberOfFields` to determine how many fields are in the specified structure `mxArray`.

Once you know the number of fields in a structure, you can loop through every field in order to set or to get field values.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisclass.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

`mxGetField`, `mxIsStruct`, `mxSetField`

Purpose

Number of elements in `ir`, `pr`, and `pi` arrays

C Syntax

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

Fortran Syntax

```
mwSize mxGetNzmax(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a sparse `mxArray`

Returns

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 software 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 the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxgetnzmax.c`
- `mxsetnzmax.c`

See Also

`mxSetNzmax`

mxGetPi (C and Fortran)

Purpose

Imaginary data elements in `mxArray` of type `double`

C Syntax

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

Fortran Syntax

```
mwPointer mxGetPi(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray` of type `double`

Returns

Pointer to the imaginary data elements of the specified `mxArray`, on success. Returns `NULL` in C (`0` in Fortran) if there is no imaginary data or if there is an error.

Description

Use `mxGetPi` on arrays of type `double` only. Use `mxIsDouble` to validate the `mxArray` type. For other `mxArray` types, use `mxGetImagData`.

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 whether an `mxArray` is purely real is to call `mxIsComplex`.

If any of the input matrices to a function are complex, MATLAB allocates the imaginary parts of all input matrices.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `convec.c`
- `findnz.c`
- `fulltosparse.c`

For Fortran examples, see:

- convec.F

See the following examples in *matlabroot*/extern/examples/mx.

- mxcalcsinglesubscript.c
- mxgetinf.c
- mxisfinite.c
- mxsetnzmax.c

See the following examples in *matlabroot*/extern/examples/mex.

- explore.c
- mexcallmatlab.c

See Also

mxGetPr, mxSetPi, mxSetPr, mxGetImagData, mxIsDouble

mxGetPr (C and Fortran)

Purpose

Real data elements in `mxArray` of type `double`

C Syntax

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

Fortran Syntax

```
mwPointer mxGetPr(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray` of type `double`

Returns

Pointer to the first element of the real data. Returns `NULL` in C (`0` in Fortran) if there is no real data.

Description

Use `mxGetPr` on arrays of type `double` only. Use `mxIsDouble` to validate the `mxArray` type. For other `mxArray` types, use `mxGetData`.

Call `mxGetPr` to access the real data in the `mxArray` that `pm` points to. Once you have the starting address, you can access any other element in the `mxArray`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillGetPrDynamicData.c`
- `arrayFillGetPr.c`
- `convec.c`
- `doubleelement.c`
- `findnz.c`
- `fulltosparse.c`
- `matrixDivide.c`
- `matrixMultiply.c`
- `sincall.c`

- timestwo.c
- timestwoalt.c
- xtimesy.c

For Fortran examples, see:

- convec.F
- dblmat.F
- fulltosparse.F
- matsq.F
- sincall.F
- timestwo.F
- xtimesy.F

See Also

mxGetPi, mxSetPi, mxSetPr, mxGetData, mxIsDouble

mxGetProperty (C and Fortran)

Purpose

Value of public property of MATLAB object

C Syntax

```
#include "matrix.h"
mxArray *mxGetProperty(const mxArray *pa, mwIndex index,
         const char *propname);
```

Fortran Syntax

```
mwPointer mxGetProperty(pa, index, propname)
mwPointer pa
mwIndex index
character*(*) propname
```

Arguments

`pa`

Pointer to an `mxArray` which is an object.

`index`

Index of the desired element of the object array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`propname`

Name of the property whose value you want to extract.

Returns

Pointer to the `mxArray` of the specified `propname` on success. Returns `NULL` in C (`0` in Fortran) if unsuccessful. Common causes of failure include:

- Specifying a nonexistent `propname`.
- Specifying a nonpublic `propname`.
- Specifying a COM or Java object.
- Specifying an `index` to an element outside the bounds of the `mxArray`. Use `mxGetNumberOfElements` or `mxGetM` and `mxGetN` to test the index value.

- Insufficient memory (in the heap) to hold the returned `mxArray`.

Description

Call `mxGetProperty` to get the value held in the specified element. In pseudo-C terminology, `mxGetProperty` returns the value at:

```
pa[index].propname
```

See Also

`mxSetProperty`, `mxGetNumberOfElements`, `mxGetM`, `mxGetN`

mxGetScalar (C and Fortran)

Purpose

Real component of first data element in `mxArray`

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to an `mxArray`; cannot be a cell `mxArray`, a structure `mxArray`, or an empty `mxArray`.

Returns

Pointer to the value of the first real (nonimaginary) element of the `mxArray`.

In C, `mxGetScalar` returns a `double`. If real elements in the `mxArray` are of a type other than `double`, `mxGetScalar` automatically converts the scalar value into a `double`. To preserve the original data representation of the scalar, cast the return value to the desired data type.

If `pm` points to a sparse `mxArray`, `mxGetScalar` returns the value of the first nonzero real element in the `mxArray`. If there are no nonzero elements, `mxGetScalar` returns `0`.

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. For example, if `pm` points to a two-dimensional `mxArray`, `mxGetScalar` returns the value of the `(1,1)` element. If `pm` points to a three-dimensional `mxArray`, `mxGetScalar` returns the value of the `(1,1,1)` element; and so on.

Use `mxGetScalar` only on a 32-bit, nonempty, numeric, logical, or char `mxArray`. Use MX Matrix Library functions such as `mxIsEmpty`,

mxIsLogical, mxIsNumeric, or mxIsChar to test for this condition before calling mxGetScalar.

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- timestwoalt.c
- xtimesy.c

See the following examples in *matlabroot*/extern/examples/mex.

- mexlock.c
- mexlockf.F
- mexsettrapflag.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetdimensions.c

See Also

mxGetM, mxGetN

mxGetString (C and Fortran)

Purpose

Copy string mxArray to C-style string

C Syntax

```
#include "matrix.h"
int mxGetString(const mxArray *pm, char *str, mwSize strlen);
```

Fortran Syntax

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

Arguments

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

str
: Starting location for the string. mxGetString writes the character data into str and then, in C, terminates the string with a NULL character (in the manner of C strings). str can point to either dynamic or static memory.

strlen
: Maximum number of characters to read into str. Typically, in C, you set strlen to 1 plus the number of elements in the string mxArray to which pm points. See the mxGetM and mxGetN reference pages to find out how to get the number of elements.

Returns

0 on success, and 1 on failure. Possible reasons for failure include

- mxArray is not a string array.
- strlen is not large enough to store the entire mxArray. If so, the function returns 1 and truncates the string.

Description

Call mxGetString to copy the character data of a string mxArray into a C-style string in C or a character array in Fortran. The copied string starts at str and contains no more than strlen-1 characters in C (no more than strlen characters in Fortran). In C, the C-style string is always terminated with a NULL character.

If the string array contains several rows, the function copies them into one long string array, one column at a time.

Multibyte Character Sets

Use this function only with strings that represent single-byte character sets. For strings that represent multibyte character sets, use the C function `mxArrayToString`. Fortran users must allocate sufficient space for the return string to avoid possible truncation.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxmalloc.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `revord.F`

See Also

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

mxIsCell (C and Fortran)

Purpose

Determine whether input is cell `mxArray`

C Syntax

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

Fortran Syntax

```
integer*4 mxIsCell(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to an array having the class `mxCELL_CLASS`, and logical `0` (`false`) otherwise.

Description

Use `mxIsCell` to determine whether the specified array is a cell array.

In C, calling `mxIsCell` is equivalent to calling:

```
mxGetClassID(pm) == mxCELL_CLASS
```

In Fortran, calling `mxIsCell` is equivalent to calling:

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

Note `mxIsCell` does not answer the question "Is this `mxArray` a cell of a cell array?" An individual cell of a cell array can be of any type.

See Also

`mxIsClass`

Purpose

Determine whether input is string mxArray

C Syntax

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

Fortran Syntax

```
integer*4 mxIsChar(pm)
mwPointer pm
```

Arguments

pm
: Pointer to an mxArray

Returns

Logical 1 (true) if pm points to an array having the class mxCHAR_CLASS, and logical 0 (false) otherwise.

Description

Use mxIsChar to determine whether pm points to string mxArray.

In C, calling mxIsChar is equivalent to calling:

```
mxGetClassID(pm) == mxCHAR_CLASS
```

In Fortran, calling mxIsChar is equivalent to calling:

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

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- phonebook.c
- revord.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatecharmatrixfromstr.c
- mxislogical.c
- mxmalloc.c

See Also mxIsClass, mxGetClassID

Purpose

Determine whether mxArray is member of specified class

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to an mxArray

`classname`

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

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

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

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

Returns

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

Description

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

In C:

```
mxIsClass(pm, "double");
```

is equivalent to calling either of these forms:

```
mxIsDouble(pm);
```

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

In Fortran:

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

is equivalent to calling either one of the following:

```
mxIsDouble(pm)
```

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisclass.c`

See Also

`mxClassID`, `mxGetClassID`, `mxIsEmpty`, `mxGetClassName`

mxIsComplex (C and Fortran)

Purpose

Determine whether data is complex

C Syntax

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

Fortran Syntax

```
integer*4 mxIsComplex(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` is a numeric array containing complex data, and logical `0` (`false`) otherwise. If `pm` points to a cell array or a structure array, `mxIsComplex` returns `false`.

Description

Use `mxIsComplex` to determine whether an imaginary part is allocated for an `mxArray`. The imaginary pointer `pi` is `NULL` in C (`0` in Fortran) if an `mxArray` is purely real and does not have any imaginary data. If an `mxArray` is complex, `pi` points to an array of numbers.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`
- `mxgetinf.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `convec.c`
- `convec.F`
- `fulltosparse.F`
- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- explore.c
- yprime.c
- mexlock.c

See Also

mxIsNumeric

mxIsDouble (C and Fortran)

Purpose

Determine whether `mxArray` represents data as double-precision, floating-point numbers

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` stores its data as double-precision, floating-point numbers, and logical `0` (`false`) otherwise.

Description

Call `mxIsDouble` to determine whether the specified `mxArray` represents its real and imaginary data as double-precision, floating-point numbers.

Older versions of MATLAB software store all `mxArray` data as double-precision, floating-point numbers. However, starting with MATLAB Version 5 software, MATLAB can store real and imaginary data in various numerical formats.

In C, calling `mxIsDouble` is equivalent to calling:

```
mxGetClassID(pm) == mxDOUBLE_CLASS
```

In Fortran, calling `mxIsDouble` is equivalent to calling:

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `fulltosparse.c`
- `fulltosparse.F`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxgeteps.c`
- `mxgetepsf.F`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `mexget.c`

See Also

`mxIsClass`, `mxGetClassID`

mxIsEmpty (C and Fortran)

Purpose

Determine whether `mxArray` is empty

C Syntax

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

Fortran Syntax

```
integer*4 mxIsEmpty(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` is empty, and logical `0` (`false`) otherwise.

Description

Use `mxIsEmpty` to determine whether an `mxArray` contains no data. An `mxArray` is empty if the size of any of its dimensions is 0.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See Also

`mxIsClass`

Purpose

Determine whether input is finite

C Syntax

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

Fortran Syntax

```
integer*4 mxIsFinite(value)
real*8 value
```

Arguments

`value`
Double-precision, floating-point number you are testing

Returns

Logical `1` (`true`) if `value` is finite, and logical `0` (`false`) otherwise.

Description

Call `mxIsFinite` to determine whether `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See Also

`mxIsInf`, `mxIsNan`

mxIsFromGlobalWS (C and Fortran)

Purpose

Determine whether `mxArray` was copied from MATLAB global workspace

C Syntax

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

Fortran Syntax

```
integer*4 mxIsFromGlobalWS(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the array was copied out of the global workspace, and logical `0` (`false`) otherwise.

Description

`mxIsFromGlobalWS` is useful for standalone MAT-file programs. `mexIsGlobal` tells you whether the pointer you pass actually points into the global workspace.

Examples

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matcreat.c`
- `matdgns.c`

See Also

`mexIsGlobal`

Purpose

Determine whether input is infinite

C Syntax

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

Fortran Syntax

```
integer*4 mxIsInf(value)
real*8 value
```

Arguments

`value`
Double-precision, floating-point number you are testing

Returns

Logical `1` (`true`) if value is infinite, and logical `0` (`false`) otherwise.

Description

Call `mxIsInf` to determine whether `value` is equal to infinity or minus infinity. MATLAB software stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable `Inf` is built into the system; you cannot modify it.

Operations that return infinity include

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See Also

`mxIsFinite`, `mxIsNaN`

mxIsInt16 (C and Fortran)

Purpose

Determine whether mxArray represents data as signed 16-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsInt16 to determine whether the specified array represents its real and imaginary data as 16-bit signed integers.

In C, calling mxIsInt16 is equivalent to calling:

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

In Fortran, calling mxIsInt16 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

Purpose

Determine whether mxArray represents data as signed 32-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsInt32 to determine whether the specified array represents its real and imaginary data as 32-bit signed integers.

In C, calling mxIsInt32 is equivalent to calling:

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

In Fortran, calling mxIsInt32 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

mxIsInt64 (C and Fortran)

Purpose

Determine whether mxArray represents data as signed 64-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsInt64 to determine whether the specified array represents its real and imaginary data as 64-bit signed integers.

In C, calling mxIsInt64 is equivalent to calling:

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

In Fortran, calling mxIsInt64 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

Purpose

Determine whether mxArray represents data as signed 8-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsInt8 to determine whether the specified array represents its real and imaginary data as 8-bit signed integers.

In C, calling mxIsInt8 is equivalent to calling:

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

In Fortran, calling mxIsInt8 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

mxIsLogical (C and Fortran)

Purpose

Determine whether `mxArray` is of type `mxLogical`

C Syntax

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

Fortran Syntax

```
integer*4 mxIsLogical(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to a logical `mxArray`, and logical `0` (`false`) otherwise.

Description

Use `mxIsLogical` to determine whether MATLAB software treats the data in the `mxArray` as Boolean (logical). If an `mxArray` is logical, MATLAB treats all zeros as meaning `false` and all nonzero values as meaning `true`. For additional information on the use of logical variables in MATLAB software, type `help logical` at the MATLAB prompt.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxislogical.c`

See Also

`mxIsClass`

Purpose

Determine whether scalar `mxArray` is of type `mxLogical`

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` is of class `mxLogical` and has `1`-by-`1` dimensions, and logical `0` (`false`) otherwise.

Description

Use `mxIsLogicalScalar` to determine whether MATLAB software treats the scalar data in the `mxArray` as logical or numerical. For additional information on the use of logical variables in MATLAB software, type `help logical` at the MATLAB prompt.

`mxIsLogicalScalar(pa)` is equivalent to:

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1
```

See Also

`mxIsLogical`, `mxIsLogicalScalarTrue`, `mxGetLogicals`, `mxGetScalar`

mxIsLogicalScalarTrue (C)

Purpose

Determine whether scalar `mxArray` of type `mxLogical` is `true`

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the value of the `mxArray`'s logical, scalar element is `true`, and logical `0` (`false`) otherwise.

Description

Use `mxIsLogicalScalarTrue` to determine whether the value of a scalar `mxArray` is true or false. For additional information on the use of logical variables in MATLAB software, type `help logical` at the MATLAB prompt.

`mxIsLogicalScalarTrue(pa)` is equivalent to:

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1 &&
mxGetLogicals(pa)[0] == true
```

See Also

`mxIsLogical`, `mxIsLogicalScalar`, `mxGetLogicals`, `mxGetScalar`

Purpose

Determine whether input is NaN (Not-a-Number)

C Syntax

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

Fortran Syntax

```
integer*4 mxIsNaN(value)
real*8 value
```

Arguments

`value`
Double-precision, floating-point number you are testing

Returns

Logical `1` (`true`) if `value` is `NaN` (Not-a-Number), and logical `0` (`false`) otherwise.

Description

Call `mxIsNaN` to determine whether `value` is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. A `NaN` is obtained as a result of mathematically undefined operations such as

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

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value; rather, it is a family of numbers that MATLAB software (and other IEEE-compliant applications) uses to represent an error condition or missing data.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `findnz.c`
- `fulltosparse.c`

See Also `mxIsFinite`, `mxIsInf`

Purpose

Determine whether `mxArray` is numeric

C Syntax

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

Fortran Syntax

```
integer*4 mxIsNumeric(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the array can contain numeric data. The following class IDs represent storage types for arrays that can contain numeric data:

- `mxDOUBLE_CLASS`
- `mxSINGLE_CLASS`
- `mxINT8_CLASS`
- `mxUINT8_CLASS`
- `mxINT16_CLASS`
- `mxUINT16_CLASS`
- `mxINT32_CLASS`
- `mxUINT32_CLASS`
- `mxINT64_CLASS`
- `mxUINT64_CLASS`

Logical `0` (`false`) if the array cannot contain numeric data.

Description

Call `mxIsNumeric` to determine whether the specified array contains numeric data. If the specified array has a storage type that represents

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

Call `mxGetClassID` to determine the exact storage type.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See the following examples in *`matlabroot`*`/extern/examples/eng_mat`.

- `matdemo1.F`

See Also

`mxGetClassID`

Purpose

Determine whether `mxArray` represents data as single-precision, floating-point numbers

C Syntax

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

Fortran Syntax

```
integer*4 mxIsSingle(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the array stores its data as single-precision, floating-point numbers, and logical `0` (`false`) otherwise.

Description

Use `mxIsSingle` to determine whether the specified array represents its real and imaginary data as single-precision, floating-point numbers.

In C, calling `mxIsSingle` is equivalent to calling:

```
mxGetClassID(pm) == mxSINGLE_CLASS
```

In Fortran, calling `mxIsSingle` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`

mxIsSparse (C and Fortran)

Purpose

Determine whether input is sparse mxArray

C Syntax

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

Fortran Syntax

```
integer*4 mxIsSparse(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if pm points to a sparse mxArray, and logical 0 (false) otherwise. A false return value means that pm points to a full mxArray or that pm does not point to a legal mxArray.

Description

Use mxIsSparse to determine whether pm points to a sparse mxArray. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- phonebook.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxgetnzmax.c
- mxsetdimensions.c
- mxsetdimensionsf.F
- mxsetnzmax.c

See Also

mxGetIr, mxGetJc, mxCreateSparse

Purpose

Determine whether input is structure `mxArray`

C Syntax

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

Fortran Syntax

```
integer*4 mxIsStruct(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to a structure `mxArray`, and logical `0` (`false`) otherwise.

Description

Use `mxIsStruct` to determine whether `pm` points to a structure `mxArray`. Many routines (for example, `mxGetFieldName` and `mxSetField`) require a structure `mxArray` as an argument.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `phonebook.c`

See Also

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

mxIsUint16 (C and Fortran)

Purpose

Determine whether mxArray represents data as unsigned 16-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsUint16 to determine whether the specified mxArray represents its real and imaginary data as 16-bit unsigned integers.

In C, calling mxIsUint16 is equivalent to calling:

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

In Fortran, calling mxIsUint16 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint32, mxIsUint64

Purpose

Determine whether mxArray represents data as unsigned 32-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsUint32 to determine whether the specified mxArray represents its real and imaginary data as 32-bit unsigned integers.

In C, calling mxIsUint32 is equivalent to calling:

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

In Fortran, calling mxIsUint32 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint64

mxIsUint64 (C and Fortran)

Purpose

Determine whether mxArray represents data as unsigned 64-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsUint64 to determine whether the specified mxArray represents its real and imaginary data as 64-bit unsigned integers.

In C, calling mxIsUint64 is equivalent to calling:

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

In Fortran, calling mxIsUint64 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32

Purpose

Determine whether mxArray represents data as unsigned 8-bit integers

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to an mxArray

Returns

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

Description

Use mxIsUint8 to determine whether the specified mxArray represents its real and imaginary data as 8-bit unsigned integers.

In C, calling mxIsUint8 is equivalent to calling:

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

In Fortran, calling mxIsUint8 is equivalent to calling:

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

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint16, mxIsUint32, mxIsUint64

mxLogical (C)

Purpose

Type for logical `mxArray`

Description

All logical `mxArray`s store their data elements as `mxLogical` rather than as `bool`.

The header file containing this type is:

```
#include "matrix.h"
```

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxislogical.c`

See Also

`mxCreateLogicalArray`

Purpose

Allocate dynamic memory using MATLAB memory manager

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxMalloc(mwSize n);
```

Fortran Syntax

```
mwPointer mxMalloc(n)
mwSize n
```

Arguments

`n`

Number of bytes to allocate

Returns

Pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a standalone (non-MEX-file) application, `mxMalloc` returns `NULL` in C (`0` in Fortran). 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

Call `mxMalloc` in MATLAB applications instead of the ANSI C `malloc` function to allocate memory. In standalone applications, such as MATLAB engine, `mxMalloc` calls the `malloc` function. In MEX-files, `mxMalloc` automatically:

- Allocates enough contiguous heap space to hold `n` bytes.
- Registers the returned heap space with the MATLAB memory manager.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using the `mxSetPr` function, MATLAB is responsible for freeing the memory.

If you use the data internally, the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt.

In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling this function. 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.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxmalloc.c`
- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillSetPr.c`

See Also

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

Purpose

Reallocate memory

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxRealloc(void *ptr, mwSize size);
```

Fortran Syntax

```
mwPointer mxRealloc(ptr, size)
mwPointer ptr
mwSize size
```

Arguments

ptr
: Pointer to a block of memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`

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

Returns

Pointer to the reallocated block of memory, or `NULL` in C (`0` in Fortran) if `size` is `0`. In a standalone (non-MEX-file) application, if not enough memory is available to expand the block to the given size, `mxRealloc` returns `NULL` in C (`0` in Fortran). In a MEX-file, if memory is not available, the MEX-file terminates and returns control to the MATLAB prompt.

Description

`mxRealloc` changes the size of a memory block that has been allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc`.

If `size` is `0` and `ptr` is not `NULL` in C (`0` in Fortran), `mxRealloc` frees the memory pointed to by `ptr` and returns `NULL` in C (`0` in Fortran).

If `size` is greater than `0` and `ptr` is `NULL` in C (`0` in Fortran), `mxRealloc` behaves like `mxMalloc`, allocating a new block of memory of `size` bytes and returning a pointer to the new block.

Otherwise, `mxRealloc` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents of the reallocated memory are unchanged up to the smaller of the new and old sizes. The reallocated memory might be in a different location from the original memory, so

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

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

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using the `mxSetPr` function, MATLAB is responsible for freeing the memory.

If you use the data internally, the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling this function. 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.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetnzmax.c`

See Also

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

Purpose

Remove field from structure array

C Syntax

```
#include "matrix.h"
void mxRemoveField(mxArray *pm, int fieldnumber);
```

Fortran Syntax

```
subroutine mxRemoveField(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

Number of the field you want to remove. In C, to remove the first field, set `fieldnumber` to `0`; to remove the second field, set `fieldnumber` to `1`; and so on. In Fortran, to remove the first field, set `fieldnumber` to `1`; to remove the second field, set `fieldnumber` to `2`; and so on.

Description

Call `mxRemoveField` to remove a field from a structure array. If the field does not exist, nothing happens. This function does not destroy the field values. Use `mxDestroyArray` to destroy the actual field values.

Consider a MATLAB structure initialized to:

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

In C, the field number `0` represents the field `name`; field number `1` represents field `billing`; field number `2` represents field `test`. In Fortran, the field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`.

See Also

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

mxSetCell (C and Fortran)

Purpose

Set value of one cell of `mxArray`

C Syntax

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

Fortran Syntax

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

Arguments

`pm`
: Pointer to a cell `mxArray`

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

`value`
: Pointer to new value for the cell. You can put an `mxArray` of any type into a cell. You can even put another cell `mxArray` into a cell.

Description

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

Note Inputs to a MEX-file are constant read-only `mxArray`s. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

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

Examples

See the following examples in *matlabroot*/extern/examples/refbook.

- phonebook.c

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatecellmatrix.c
- mxcreatecellmatrixf.F

See Also

mxCreateCellArray, mxCreateCellMatrix, mxGetCell, mxIsCell, mxDestroyArray

mxSetClassName (C)

Purpose

Convert structure array to MATLAB object array

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray` of class `mxSTRUCT_CLASS`

`classname`
Object class to which to convert `array_ptr`

Returns

`0` if successful, and nonzero otherwise. One cause of failure is that `array_ptr` is not a structure `mxArray`. Call `mxIsStruct` to determine whether `array_ptr` is a structure.

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 software 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 *pm, void *pr);
```

Fortran Syntax

```
subroutine mxSetData(pm, pr)
mwPointer pm, pr
```

Arguments

`pm`

Pointer to an `mxArray`

`pr`

Pointer to an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this memory.

Description

`mxSetData` is like `mxSetPr`, except that in C, its second argument is a `void *`. Use this function on numeric arrays with contents other than `double`.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillSetData.c`

See Also

`mxCalloc`, `mxFree`, `mxGetData`, `mxSetPr`

mxSetDimensions (C and Fortran)

Purpose

Modify number of dimensions and size of each dimension

C Syntax

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

Fortran Syntax

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

Arguments

`pm`
: Pointer to an `mxArray`

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

`ndim`
: 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 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 like `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`, enlarge the `pr` (and `pi`, if it exists) arrays accordingly.

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

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array has the dimensions 4-by-1-by-7.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetdimensions.c`
- `mxsetdimensionsf.F`

See Also

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

mxSetField (C and Fortran)

Purpose

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

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

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

`index`

Index of an element in the array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for details on calculating an index.

`fieldname`

Name of a field in the structure. The field must exist in the structure. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to determine existing field names.

`pvalue`

Pointer to an `mxArray` containing the data you want to assign to `fieldname`.

Description

Use `mxSetField` to assign the contents of `pvalue` to the field `fieldname` of element `index`.

If you want to replace the contents of `fieldname`, you must first free the memory of the existing data. Use the `mxGetField` function to get a pointer to the field, call `mxDestroyArray` on the pointer, then call `mxSetField` to assign the new value.

You cannot assign `pvalue` to more than one field in a structure or to more than one element in the `mxArray`. If you want to assign the contents of `pvalue` to multiple fields, use the `mxDuplicateArray` function to make copies of the data then call `mxSetField` on each copy.

To free memory for structures created using this function, call `mxDestroyArray` only on the structure array. Do not call `mxDestroyArray` on the array `pvalue` points to. If you do, MATLAB attempts to free the same memory twice, which can corrupt memory.

Note Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

Alternatives

C Language

In C, you can replace the statements:

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

with a call to `mxSetField`:

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

Fortran Language

In Fortran, you can replace the statements:

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

with a call to mxSetField:

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatestructarray.c

See Also

mxCreateStructArray, mxCreateStructMatrix, mxGetField, mxGetFieldNameByNumber, mxGetFieldNumber, mxGetNumberOfFields, mxIsStruct, mxSetFieldByNumber, mxDestroyArray, mxCalcSingleSubscript

Purpose

Set structure array field, given field number and index

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

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

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for details on calculating an index.

`fieldnumber`

Position of the field in the structure. The field must exist in the structure.

In C, the first field within each element has a `fieldnumber` of `0`. The `fieldnumber` of the last is `N-1`, where `N` is the number of fields.

In Fortran, the first field within each element has a `fieldnumber` of `1`. The `fieldnumber` of the last is `N`, where `N` is the number of fields.

`pvalue`
Pointer to the `mxArray` containing the data you want to assign.

Description

Use `mxSetFieldByNumber` to assign the contents of `pvalue` to the field specified by `fieldnumber` of element `index`. `mxSetFieldByNumber` is like `mxSetField`; however, the function identifies the field by position number, not by name.

If you want to replace the contents at `fieldnumber`, you must first free the memory of the existing data. Use the `mxGetFieldByNumber` function to get a pointer to the field, call `mxDestroyArray` on the pointer, then call `mxSetFieldByNumber` to assign the new value.

You cannot assign `pvalue` to more than one field in a structure or to more than one element in the `mxArray`. If you want to assign the contents of `pvalue` to multiple fields, use the `mxDuplicateArray` function to make copies of the data then call `mxSetFieldByNumber` on each copy.

To free memory for structures created using this function, call `mxDestroyArray` only on the structure array. Do not call `mxDestroyArray` on the array `pvalue` points to. If you do, MATLAB attempts to free the same memory twice, which can corrupt memory.

Note Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

Alternatives

C Language

In C, calling:

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

is equivalent to calling:

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

Fortran Language

In Fortran, calling:

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

is equivalent to calling:

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxcreatestructarray.c

See Also

mxCreateStructArray, mxCreateStructMatrix, mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber, mxGetNumberOfFields, mxIsStruct, mxSetField, mxDestroyArray, mxCalcSingleSubscript

mxSetImagData (C and Fortran)

Purpose

Set imaginary data pointer for `mxArray`

C Syntax

```
#include "matrix.h"
void mxSetImagData(mxArray *pm, void *pi);
```

Fortran Syntax

```
subroutine mxSetImagData(pm, pi)
mwPointer pm, pi
```

Arguments

`pm`

Pointer to an `mxArray`

`pi`

Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory. If `pi` points to static memory, memory errors will result when the array is destroyed.

Description

`mxSetImagData` is like `mxSetPi`, except that in C, its `pi` argument is a `void *`. Use this function on numeric arrays with contents other than `double`.

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See Also

`mxCalloc`, `mxFree`, `mxGetImagData`, `mxSetPi`

Purpose

Set ir array of sparse mxArray

C Syntax

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

Fortran Syntax

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

Arguments

pm

Pointer to a sparse mxArray

ir

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

Description

Use mxSetIr to specify the ir array of a sparse mxArray. The ir array is an array of integers; the length of the ir array equals the value of nzmax.

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

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

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

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

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

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

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

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

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetnzmax.c

See the following examples in *matlabroot*/extern/examples/mex.

- explore.c

See Also

mxCreateSparse, mxGetIr, mxGetJc, mxSetJc, mxFree

mxSetJc (C and Fortran)

Purpose

Set jc array of sparse mxArray

C Syntax

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

Fortran Syntax

```
subroutine mxSetJc(pm, jc)
mwPointer pm, jc
```

Arguments

pm
: Pointer to a sparse mxArray

jc
: Pointer to the jc array

Description

Use mxSetJc to specify a new jc array for a sparse mxArray. The jc array is an integer array having n+1 elements, where n is the number of columns in the sparse mxArray.

If the jth column of the sparse mxArray has any nonzero elements:

- jc[j] is the index in ir, pr, and pi (if it exists) of the first nonzero element in the jth column.
- jc[j+1]-1 is the index of the last nonzero element in the jth column.
- For the jth column of the sparse matrix, jc[j] is the total number of nonzero elements in all preceding columns.

The number of nonzero elements in the jth column of the sparse mxArray is:

```
jc[j+1] - jc[j];
```

For the jth column of the sparse mxArray, jc[j] is the total number of nonzero elements in all preceding columns. The last element of the jc array, jc[number of columns], is equal to nnz, which is the number of nonzero elements in the entire sparse mxArray.

For example, consider a 7-by-3 sparse `mxArray` named `Sparrow` containing six nonzero elements, created by typing:

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

The following table lists the contents of the `ir`, `jc`, and `pr` arrays.

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

As an example of a much sparser `mxArray`, consider a 1000-by-8 sparse `mxArray` named `Spacious` containing only three nonzero elements. The `ir`, `pr`, and `jc` arrays contain the values listed in this table.

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

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

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- explore.c

See Also

mxCreateSparse, mxGetIr, mxGetJc, mxSetIr, mxFree

mxSetM (C and Fortran)

Purpose

Set number of rows in mxArray

C Syntax

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

Fortran Syntax

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

Arguments

pm
: Pointer to an mxArray

m
: Number of rows

Description

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

You typically use mxSetM to change the shape of an existing mxArray. The mxSetM function 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, enlarge the pr, pi, ir, and/or jc arrays. Call mxRealloc to enlarge them.

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetdimensions.c

See the following examples in *matlabroot*/extern/examples/refbook.

- sincall.c

- sincall.F

See Also

mxGetM, mxGetN, mxSetN

mxSetN (C and Fortran)

Purpose

Set number of columns in mxArray

C Syntax

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

Fortran Syntax

```
subroutine mxSetN(pm, n)
mwPointer pm
mwSize n
```

Arguments

pm
: Pointer to an mxArray

n
: 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 pm 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. The mxSetN function 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, 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, you might want to reduce the sizes of the pr, pi, ir, and/or jc arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetdimensions.c

See the following examples in *matlabroot*/extern/examples/refbook.

- sincall.c
- sincall.F

See Also

mxGetM, mxGetN, mxSetM

mxSetNzmax (C and Fortran)

Purpose

Set storage space for nonzero elements

C Syntax

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

Fortran Syntax

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

Arguments

`pm`

Pointer to a sparse `mxArray`.

`nzmax`

Number of elements `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 number of nonzero elements in the sparse `mxArray`.

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

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

Ways to determine how large to make `nzmax` are:

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetnzmax.c

See Also

mxGetNzmax, mxRealloc

mxSetPi (C and Fortran)

Purpose

Set new imaginary data for mxArray

C Syntax

```
#include "matrix.h"
void mxSetPi(mxArray *pm, double *pi);
```

Fortran Syntax

```
subroutine mxSetPi(pm, pi)
mwPointer pm, pi
```

Arguments

pm

Pointer to a full (nonsparse) mxArray

pi

Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pi points to static memory, memory leaks and other memory errors might 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 in C (1 in Fortran) or by setting pi to a non-NULL value in C (a nonzero value in Fortran)—you do not ordinarily use mxSetPi to initialize the created mxArray's imaginary elements. Rather, you call mxSetPi to replace the initial imaginary values with new ones.

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxisfinite.c
- mxsetnzmax.c

See Also mxGetPi, mxGetPr, mxSetImagData, mxSetPr, mxFree

mxSetPr (C and Fortran)

Purpose

Set new real data for mxArray

C Syntax

```
#include "matrix.h"
void mxSetPr(mxArray *pm, double *pr);
```

Fortran Syntax

```
subroutine mxSetPr(pm, pr)
mwPointer pm, pr
```

Arguments

pm

Pointer to a full (nonsparse) mxArray

pr

Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pr points to static memory, memory leaks and other memory errors can result.

Description

Use mxSetPr to set the real data of the specified mxArray.

All mxCreate* calls allocate heap space to hold real data. Therefore, you do not ordinarily use mxSetPr to initialize the real elements of a freshly created mxArray. Rather, you call mxSetPr to replace the initial real values with new ones.

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

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetnzmax.c

See Also

mxGetPi, mxGetPr, mxSetData, mxSetPi, mxFree

Purpose

Set value of public property of MATLAB object

C Syntax

```
#include "matrix.h"
void mxSetProperty(mxArray *pa, mwIndex index,
  const char *propname, const mxArray *value);
```

Fortran Syntax

```
subroutine mxSetProperty(pa, index, propname, value)
mwPointer pa, value
mwIndex index
character*(*) propname
```

Arguments

`pa`
: Pointer to an `mxArray` which is an object.

`index`
: Index of the desired element of the object array.

  In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`propname`
: Name of the property whose value you are assigning.

value
: Pointer to the `mxArray` you are assigning.

Description

Use `mxSetProperty` to assign a `value` to the specified property. In pseudo-C terminology, `mxSetProperty` performs the assignment:

```
pa[index].propname = value;
```

`mxSetProperty` makes a copy of the value before assigning it as the new property value. This might be a concern if the property uses a large amount of memory.

See Also `mxGetProperty`

Index

O

P

S