MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

The MathWorks

External Interfaces Reference

Version 6

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`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	Mail

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

MATLAB External Interfaces Reference

© COPYRIGHT 1984 - 2002 by The MathWorks, Inc.

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

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

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

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

Printing History:			
	December 1996	First printing	
	May 1997	Online only	Revised for 5.1
	January 1998	Online only	Revised for 5.2
	January 1999	Online only	Revised for 5.3
	September 2000	Online only	Revised for 6.0
	June 2001	Online only	Revised for 6.1
	July 2002	Online only	Revised for MATLAB 6.5 (Release 13)

Contents

1 C Engine Functions

2 C MAT-File Functions

3 C MEX-Functions

4 C MX-Functions

5 Fortran Engine Functions

6 Fortran MAT-File Functions

7 Fortran MEX-Functions

8 Fortran MX-Functions

9 Java Interface Functions

10 COM Functions

11 DDE Functions

12 Serial Port I/O Functions

External Interfaces/API Reference

This section contains the MATLAB External Interfaces function reference pages. This includes reference pages for what was formerly called the MATLAB Application Program Interface, or API.

Category	Description
C Engine Functions	Functions that allow you to call MATLAB from your own C programs.
C MAT-File Functions	Functions that allow you to incorporate and use MATLAB data in your own C programs.
C MEX-Functions	Functions that you use in your C MEX-files to perform operations back in the MATLAB environment.
C MX-Functions	Array access and creation functions that you use in your C MEX-files to manipulate MATLAB arrays.
Fortran Engine Functions	Functions that allow you to call MATLAB from your own Fortran programs.
Fortran MAT-File Functions	Functions that allow you to incorporate and use MATLAB data in your own Fortran programs.
Fortran MEX-Functions	Functions that you use in your Fortran MEX-files to perform operations back in the MATLAB environment.
Fortran MX-Functions	Array access and creation functions that you use in your Fortran MEX-files to manipulate MATLAB arrays.
Java Interface Functions	Functions that enable you to create and interact with Java classes and objects from MATLAB.

Category	Description
COM Functions	Functions that create COM objects and manipulate their interfaces.
DDE Functions	Dynamic Data Exchange functions that enable MATLAB to access other Windows applications and vice versa.
Serial Port I/O Functions	Functions that enable you to interact with devices connected to your computer's serial port.

C Engine Functions

`engClose`	Quit MATLAB engine session
`engEvalString`	Evaluate expression in string
`engGetArray (Obsolete)`	Use `engGetVariable`
`engGetFull (Obsolete)`	Use `engGetVariable` followed by appropriate `mxGet` routines
`engGetMatrix (Obsolete)`	Use `engGetVariable`
`engGetVariable`	Copy variable from engine workspace
`engGetVisible`	Determine visibility of engine session
`engOpen`	Start MATLAB engine session
`engOpenSingleUse`	Start MATLAB engine session for single, nonshared use
`engOutputBuffer`	Specify buffer for MATLAB output
`engPutArray (Obsolete)`	Use `engPutVariable`
`engPutFull (Obsolete)`	Use `mxCreateDoubleMatrix` and `engPutVariable`
`engPutMatrix (Obsolete)`	Use `engPutVariable`
`engPutVariable`	Put variables into engine workspace
`engSetEvalCallback (Obsolete)`	Function is obsolete
`engSetEvalTimeout (Obsolete)`	Function is obsolete
`engSetVisible`	Show or hide engine session
`engWinInit (Obsolete)`	Function is obsolete

Purpose Quit a MATLAB engine session

C Syntax

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

Arguments

`ep`
Engine pointer.

Description This routine allows you to quit a MATLAB engine session.

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

Examples

UNIX

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

Windows

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

engEvalString

Purpose

Evaluate expression in string

C Syntax

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

Arguments

`ep`
Engine pointer.

`string`
String to execute.

Description

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

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

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

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

Examples

UNIX

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

Windows

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

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
engGetVariable
```

instead of

```
engGetArray
```

See Also

`engGetVariable`, `engPutVariable`, and examples in the `eng_mat` subdirectory of the `examples` directory

engGetFull (Obsolete)

V4 Compatible

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

Use

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

instead of

`engGetFull`

For example,

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

   pmat = engGetVariable(ep, name);

   if (!pmat)
           return(1);

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

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

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

    mxDestroyArray(pmat);

    return(0);
}
```

See Also

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

engGetMatrix (Obsolete)

V4 Compatible

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

Use

```
engGetVariable
```

instead of

```
engGetMatrix
```

See Also

`engGetVariable`, `engPutVariable`, and examples in the `eng_mat` subdirectory of the `examples` directory

Purpose

Copy a variable from a MATLAB engine's workspace

C Syntax

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

Arguments

`ep`
Engine pointer.

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

Description

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

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

Examples

UNIX

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

Windows

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

See Also

`engPutVariable`

engGetVisible

Purpose

Determine visibility of MATLAB engine session

C Syntax

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

Arguments

`ep`
Engine pointer.

`value`
Pointer to value returned from `engGetVisible`.

Description

Windows Only

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

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

Examples

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

```
Engine *ep;
bool vis;

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

To determine the current visibility setting, use

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

See Also

`engSetVisible`

Purpose

Start a MATLAB engine session

C Syntax

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

Arguments

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

Returns

A pointer to an engine handle.

Description

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

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

On UNIX systems, if `startcmd` is `NULL` or the empty string, `engOpen` starts MATLAB on the current host using the command `matlab`. If `startcmd` is a hostname, `engOpen` starts MATLAB on the designated host by embedding the specified hostname string into the larger string:

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

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

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

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

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

```
matlab /regserver
```

See “Introducing MATLAB COM Integration” for additional details.

Examples

UNIX

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

Windows

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

Purpose Start a MATLAB engine session for single, nonshared use

C Syntax

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

Arguments

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

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

`retstatus`
Return status; possible cause of failure.

Description

Windows

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

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

```
matlab /regserver
```

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

See Introducing MATLAB COM Integration for additional details.

UNIX

This routine is not supported and simply returns.

engOutputBuffer

Purpose

Specify buffer for MATLAB output

C Syntax

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

Arguments

`ep`
Engine pointer.

`n`
Length of buffer `p`.

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

Description

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

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

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

Examples

UNIX

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

Windows

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

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
engPutVariable
```

instead of

```
engPutArray
```

See Also

`engPutVariable`, `engGetVariable`, and examples in the `eng_mat` subdirectory of the `examples` directory

engPutFull (Obsolete)

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

Use

`mxCreateDoubleMatrix` and `engPutVariable`

instead of

`engPutFull`

For example,

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

   pmat = mxCreateDoubleMatrix(0, 0, mxCOMPLEX);

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

   retval = engPutVariable(ep, name, pmat);

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

   mxDestroyArray(pmat);
```

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

See Also

engGetVariable, mxCreateDoubleMatrix

engPutMatrix (Obsolete)

V4 Compatible

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

Use

```
engPutVariable
```

instead of

```
engPutMatrix
```

See Also

`engPutVariable`

Purpose

Put variables into a MATLAB engine's workspace

C Syntax

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

Arguments

`ep`
Engine pointer.

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

`mp`
`mxArray` pointer.

Description

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

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

Examples

UNIX

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

Windows

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

engSetEvalCallback (Obsolete)

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

V4 Compatible

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

engSetVisible

Purpose

Show or hide MATLAB engine session

C Syntax

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

Arguments

`ep`
Engine pointer.

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

Description

Windows Only

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

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

Examples

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

```
Engine *ep;
bool vis;

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

To determine the current visibility setting, use

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

See Also

`engGetVisible`

V4 Compatible

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

2

C MAT-File Functions

`matClose`	Close MAT-file
`matDeleteArray (Obsolete)`	Use `matDeleteVariable`
`matDeleteMatrix (Obsolete)`	Use `matDeleteVariable`
`matDeleteVariable`	Delete named `mxArray` from MAT-file
`matGetArray (Obsolete)`	Use `matGetVariable`
`matGetArrayHeader (Obsolete)`	Use `matGetVariableInfo`
`matGetDir`	Get directory of `mxArrays` in MAT-file
`matGetFp`	Get file pointer to MAT-file
`matGetFull (Obsolete)`	Use `matGetVariable` followed by the appropriate `mxGet` routines
`matGetMatrix (Obsolete)`	Use `matGetVariable`
`matGetNextArray (Obsolete)`	Use `matGetNextVariable`
`matGetNextArrayHeader (Obsolete)`	Use mat `GetNextArrayHeaderFromMATfile`
`matGetNextMatrix (Obsolete)`	Use `matGetNextVariable`
`matGetNextVariable`	Read next `mxArray` from MAT-file
`matGetNextVariableInfo`	Load array header information only
`matGetString (Obsolete)`	Use `matGetVariable` and `mxGetString`
`matGetVariable`	Read `mxArray` from MAT-file
`matGetVariableInfo`	Load header array information only
`matOpen`	Open MAT-file

`matPutArray (Obsolete)`	Use `matPutVariable`
`matPutArrayAsGlobal (Obsolete)`	Use `matPutVariableAsGlobal`
`matPutFull (Obsolete)`	Use `mxCreateDoubleMatrix` and `matPutVariable`
`matPutMatrix (Obsolete)`	Use `matPutVariable`
`matPutString (Obsolete)`	Use `mxCreateString` and `matPutVariable`
`matPutVariable`	Write `mxArrays` into MAT-files
`matPutVariableAsGlobal`	Put `mxArrays` into MAT-files

matClose

Purpose

Closes a MAT-file

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

Description

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

Examples

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

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matDeleteVariable
```

instead of

```
matDeleteArray
```

See Also

`matDeleteVariable`

matDeleteMatrix (Obsolete)

V4 Compatible

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

Use

```
matDeleteVariable
```

instead of

```
matDeleteMatrix
```

See Also

`matDeleteVariable`

Purpose

Delete named mxArray from MAT-file

C Syntax

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

Arguments

mfp
Pointer to MAT-file information.

name
Name of mxArray to delete.

Description

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

Examples

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

matGetArray (Obsolete)

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matGetVariable
```

instead of

```
matGetArray
```

See Also

`matGetVariable`

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matGetVariableInfo
```

instead of

```
matGetArrayHeader
```

See Also

`matGetVariableInfo`

matGetDir

Purpose Get directory of mxArrays in a MAT-file

C Syntax

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

Arguments

mfp
Pointer to MAT-file information.

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

Description

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

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

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

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

Examples

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

Purpose Get file pointer to a MAT-file

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

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

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

V4 Compatible

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

Use

`matGetVariable` followed by the appropriate `mxGet` routines

instead of

`matGetFull`

For example,

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

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

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

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

    mxDestroyArray(parr);
```

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

See Also matGetVariable

matGetMatrix (Obsolete)

V4 Compatible

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

Use

```
matGetVariable
```

instead of

```
matGetMatrix
```

See Also

`matGetVariable`

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matGetNextVariable
```

instead of

```
matGetNextArray
```

See Also

`matGetNextVariable`

matGetNextArrayHeader (Obsolete)

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matGetNextVariableInfo
```

instead of

```
matGetNextArrayHeader
```

See Also

`matGetNextVariableInfo`

V4 Compatible

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

Use

```
matGetNextVariable
```

instead of

```
matGetNextMatrix
```

See Also

`matGetNextVariable`

matGetNextVariable

Purpose

Read next mxArray from MAT-file

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

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

Description

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

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

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

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

Examples

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

matGetNextVariableInfo

Purpose Load array header information only

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

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

Description

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

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

Examples

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

See Also

`matGetNextVariable`, `matGetVariableInfo`

matGetString (Obsolete)

V4 Compatible

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

Use

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

instead of

```
matGetString
```

See Also

`matGetVariable`, `mxGetString`

Purpose

Read mxArrays from MAT-files

C Syntax

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

Arguments

mfp
Pointer to MAT-file information.

name
Name of mxArray to get from MAT-file.

Description

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

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

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

Examples

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

matGetVariableInfo

Purpose

Load array header information only

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray`.

Description

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

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

Examples

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

Purpose

Opens a MAT-file

C Syntax

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

Arguments

`filename`
Name of file to open.

`mfp`
Pointer to MAT-file information.

`mode`
File opening mode. Legal values for `mode` are:

Table 1-1:

`r`	Opens file for reading only; determines the current version of the MAT-file by inspecting the files and preserves the current version.
`u`	Opens file for update, both reading and writing, but does not create the file if the file does not exist (equivalent to the `r+` mode of `fopen`); determines the current version of the MAT-file by inspecting the files and preserves the current version.
`w`	Opens file for writing only; deletes previous contents, if any.
`w4`	Creates a MATLAB 4 MAT-file.

Description

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

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

Examples

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

matPutArray (Obsolete)

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matPutVariable
```

instead of

```
matPutArray
```

See Also

`matPutVariable`

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
matPutVariableAsGlobal
```

instead of

```
matPutArrayAsGlobal
```

See Also

`matPutVariableAsGlobal`

matPutFull (Obsolete)

V4 Compatible

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

Use

`mxCreateDoubleMatrix` and `matPutVariable`

instead of

`matPutFull`

For example,

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

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

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

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

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

   mxDestroyArray(parr);
```

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

See Also

`mxCreateDoubleMatrix`, `matPutVariable`

matPutMatrix (Obsolete)

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

Use

```
matPutVariable
```

instead of

```
matPutMatrix
```

See Also `matPutVariable`

V4 Compatible

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

Use

```
#include "matrix.h"
#include "mat.h"
mp = mxCreateString(str);
matPutVariable(mfp, name, mp);
mxDestroyArray(mp);
```

instead of

```
matPutString(mfp, name, str);
```

See Also

`matPutVariable`

matPutVariable

Purpose

Write mxArrays into MAT-files

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

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

`mp`
`mxArray` pointer.

Description

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

`matPutVariable` writes `mxArray mp` to the MAT-file `mfp`. If the `mxArray` does not exist in the MAT-file, it is appended to the end. If an `mxArray` with the same name already exists in the file, the existing `mxArray` is replaced with the new `mxArray` by rewriting the file. The size of the new `mxArray` can be different than the existing `mxArray`.

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

Examples

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

matPutVariableAsGlobal

Purpose

Put mxArrays into MAT-files as originating from the global workspace

C Syntax

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

Arguments

`mfp`
Pointer to MAT-file information.

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

`mp`
`mxArray` pointer.

Description

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

`matPutVariableAsGlobal` writes `mxArray mp` to the MAT-file `mfp`. If the `mxArray` does not exist in the MAT-file, it is appended to the end. If an `mxArray` with the same name already exists in the file, the existing `mxArray` is replaced with the new `mxArray` by rewriting the file. The size of the new `mxArray` can be different than the existing `mxArray`.

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

Examples

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

matPutVariableAsGlobal

3

C MEX-Functions

`mexGetMatrix (Obsolete)`	Use `mexGetVariable`
`mexGetMatrixPtr (Obsolete)`	Use `mexGetVariablePtr`
`mexGetNaN (Obsolete)`	Use `mxGetNaN`
`mexGetVariable`	Get copy of variable from another workspace
`mexGetVariablePtr`	Get read-only pointer to variable from another workspace
`mexIsFinite (Obsolete)`	Use `mxIsFinite`
`mexIsGlobal`	True if `mxArray` has global scope
`mexIsInf (Obsolete)`	Use `mxIsInf`
`mexIsLocked`	True if MEX-file is locked
`mexIsNaN (Obsolete)`	Use `mxIsNaN`
`mexLock`	Lock MEX-file so it cannot be cleared from memory
`mexMakeArrayPersistent`	Make `mxArray` persist after MEX-file completes
`mexMakeMemoryPersistent`	Make memory allocated by MATLAB memory allocation routines persist after MEX-file completes
`mexPrintf`	ANSI C `printf`-style output routine
`mexPutArray (Obsolete)`	Use `mexPutVariable`
`mexPutFull (Obsolete)`	Use `mxCreateDoubleMatrix`, `mxSetPr`, `mxSetPi`, `mexPutVariable`
`mexPutMatrix (Obsolete)`	Use `mexPutVariable`
`mexPutVariable`	Copy `mxArray` from your MEX-file into another workspace
`mexSet`	Set value of Handle Graphics property

`mexSetTrapFlag`	Control response of `mexCallMATLAB` to errors
`mexUnlock`	Unlock MEX-file so it can be cleared from memory
`mexWarnMsgIdAndTxt`	Issue warning message with identifier
`mexWarnMsgTxt`	Issue warning message

mexAddFlops (Obsolete)

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

Purpose

Register a function to be called when the MEX-function is cleared or when MATLAB terminates

C Syntax

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

Arguments

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

Returns

Always returns 0.

Description

Use `mexAtExit` to register a C function to be called just before the MEX-function is cleared or MATLAB is terminated. `mexAtExit` gives your MEX-function a chance to perform tasks such as freeing persistent memory and closing files. Typically, the named `ExitFcn` performs tasks like closing streams or sockets.

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

If a MEX-function is locked, all attempts to clear the MEX-file will fail. Consequently, if a user attempts to clear a locked MEX-file, MATLAB does not call the `ExitFcn`.

Examples

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

See Also

`mexLock`, `mexUnlock`

mexCallMATLAB

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

C Syntax

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

Arguments

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

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

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

`prhs`
Pointer to an array of input arguments.

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

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

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

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

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

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

you get this warning message in MATLAB:

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

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

Examples

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

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

See Also

`mexFunction`, `mexSetTrapFlag`

mexErrMsgIdAndTxt

Purpose

Issue error message with identifier and return to the MATLAB prompt

C Syntax

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

Arguments

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

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

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

Description

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

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

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

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

See Also

`mexErrMsgTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

Purpose Issue error message and return to the MATLAB prompt

C Syntax

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

Arguments

`error_msg`
String containing the error message to be displayed.

Description

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

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

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

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

Examples

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

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

See Also

`mexErrMsgIdAndTxt`, `mexWarnMsgTxt`, `mexWarnMsgIdAndTxt`

mexEvalString

Purpose

Execute a MATLAB command in the workspace of the caller

C Syntax

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

Arguments

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

Returns

0 if successful, and a nonzero value if unsuccessful.

Description

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

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

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

Examples

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

See Also

`mexCallMATLAB`

Purpose

Entry point to a C MEX-file

C Syntax

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

Arguments

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

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

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

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

Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the generic name of the function entry point that must exist in every C source MEX-file. When you invoke a MEX-function, MATLAB finds and loads the corresponding MEX-file of the same name. MATLAB then searches for a symbol named `mexFunction` within the MEX-file. If it finds one, it calls the MEX-function using the address of the `mexFunction` symbol. If MATLAB cannot find a routine named `mexFunction` inside the MEX-file, it issues an error message.

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

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

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

Examples

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

Purpose Gives the name of the current MEX-function

C Syntax

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

Arguments `none`

Returns The name of the current MEX-function.

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

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

mexGet

Purpose Get the value of the specified Handle Graphics® property

C Syntax

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

Arguments

`handle`
Handle to a particular graphics object.

`property`
A Handle Graphics property.

Returns The value of the specified property in the specified graphics object on success. Returns `NULL` on failure. The return argument from `mexGet` is declared as `constant`, meaning that it is read only and should not be modified. Changing the data in these `mxArrays` may produce undesired side effects.

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

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

See Also `mexSet`

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
mexGetVariable(workspace, var_name);
```

instead of

```
mexGetArray(var_name, workspace);
```

See Also

`mexGetVariable`

mexGetArrayPtr (Obsolete)

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
mexGetVariablePtr(var_name, workspace);
```

instead of

```
mexGetArrayPtr(var_name, workspace);
```

See Also

`mexGetVariable`

V4 Compatible

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

Use

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

instead of

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

See Also

`mxGetEps`

mexGetFull (Obsolete)

V4 Compatible

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

Use

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

instead of

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

See Also

`mexGetVariable`, `mxGetPr`, `mxGetPi`

V4 Compatible

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

Use

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

instead of

```
mexGetGlobal(name);
```

See Also

`mexGetVariable`, `mxGetName (Obsolete)`, `mxGetPr`, `mxGetPi`

mexGetInf (Obsolete)

V4 Compatible

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

Use

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

instead of

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

See Also

`mxGetInf`

V4 Compatible

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

Use

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

instead of

```
mexGetMatrix(name);
```

See Also

`mexGetVariable`

mexGetMatrixPtr (Obsolete)

V4 Compatible

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

Use

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

instead of

```
mexGetMatrixPtr(name);
```

See Also

`mexGetVariablePtr`

V4 Compatible

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

Use

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

instead of

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

See Also

`mxGetNaN`

mexGetVariable

Purpose

Get a copy of a variable from the specified workspace

C Syntax

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

Arguments

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

`base`	Search for the variable in the base workspace
`caller`	Search for the variable in the caller's workspace
`global`	Search for the variable in the global workspace

`var_name`
Name of the variable to copy.

Returns

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

Description

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

Examples

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

See Also

`mexGetVariablePtr`, `mexPutVariable`

Purpose

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

C Syntax

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

Arguments

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

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

`base`	Search for the variable in the base workspace
`caller`	Search for the variable in the caller's workspace
`global`	Search for the variable in the global workspace

Returns

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

Description

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

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

Examples

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

See Also

`mexGetVariable`

mexIsFinite (Obsolete)

V4 Compatible

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

Use

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

instead of

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

See Also

`mxIsFinite`

Purpose True if `mxArray` has global scope

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

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

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

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

See Also `mexGetVariable`, `mexGetVariablePtr`, `mexPutVariable`, `global`

mexIsInf (Obsolete)

V4 Compatible

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

Use

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

instead of

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

See Also

`mxIsInf`

Purpose Determine if this MEX-file is locked

C Syntax

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

Returns `True` if the MEX-file is locked; `False` if the file is unlocked.

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

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

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

See Also `mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

mexIsNaN (Obsolete)

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

Use

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

instead of

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

See Also `mxIsInf`

Purpose Lock a MEX-file so that it cannot be cleared from memory

C Syntax

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

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

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

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

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

See Also `mexIsLocked`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

mexMakeArrayPersistent

Purpose

Make an mxArray persist after the MEX-file completes

C Syntax

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

Arguments

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

Description

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

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

See Also

mexAtExit, mexLock, mexMakeMemoryPersistent, and the mxCreate functions.

Purpose

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

C Syntax

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

Arguments

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

Description

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

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

See Also

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

Purpose

ANSI C `printf`-style output routine

C Syntax

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

Arguments

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

Returns

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

Description

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

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

Examples

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

See Also

`mexErrMsgTxt`, `mexWarnMsgTxt`

V5 Compatible

This API function is obsolete and should not be used in a program that interfaces with MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

Use

```
mexPutVariable(workspace, var_name, array_ptr);
```

instead of

```
mexPutArray(array_ptr, workspace);
```

See Also

`mexPutVariable`

mexPutFull (Obsolete)

V4 Compatible

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

Use

```
array_ptr = mxCreateDoubleMatrix(m, n, mxREAL/mxCOMPLEX);
mxSetPr(array_ptr, pr);
mxSetPi(array_ptr, pi);
mexPutVariable("caller", name, array_ptr);
```

instead of

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

See Also

`mxSetM`, `mxSetN`, `mxSetPr`, `mxSetPi`, `mexPutVariable`

V4 Compatible

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

Use

```
mexPutVariable("caller", var_name, array_ptr);
```

instead of

```
mexPutMatrix(matrix_ptr);
```

See Also

`mexPutVariable`

mexPutVariable

Purpose

Copy an mxArray from your MEX-function into the specified workspace

C Syntax

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

Arguments

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

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

var_name
Name given to the mxArray in the workspace.

array_ptr
Pointer to the mxArray.

Returns

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

Description

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

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

If a variable of the same name already exists in the specified workspace, mexPutVariable overwrites the previous contents of the variable with the contents of the new mxArray. For example, suppose the MATLAB workspace defines variable Peaches as

```
Peaches
1     2     3     4
```

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

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

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

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

See Also `mexGetVariable`

mexSet

Purpose

Set the value of the specified Handle Graphics property

C Syntax

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

Arguments

`handle`
Handle to a particular graphics object.

`property`
String naming a Handle Graphics property.

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

Returns

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

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

Description

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

Examples

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

See Also

`mexGet`

Purpose

Control response of `mexCallMATLAB` to errors

C Syntax

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

Arguments

`trap_flag`
Control flag. Currently, the only legal values are:

`0` On error, control returns to the MATLAB prompt.

`1` On error, control returns to your MEX-file.

Description

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

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

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

Examples

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

See Also

`mexAtExit`, `mexErrMsgTxt`

mexUnlock

Purpose Unlock this MEX-file so that it can be cleared from memory

C Syntax

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

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

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

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

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

Purpose

Issue warning message with identifier

C Syntax

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

Arguments

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

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

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

Description

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

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

See Also

`mexWarnMsgTxt`, `mexErrMsgIdAndTxt`, `mexErrMsgTxt`

mexWarnMsgTxt

Purpose

Issue warning message

C Syntax

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

Arguments

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

Description

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

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

Examples

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

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

See Also

`mexWarnMsgIdAndTxt`, `mexErrMsgTxt`, `mexErrMsgIdAndTxt`

C MX-Functions

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

`mxCreateDoubleMatrix`	Create unpopulated two-dimensional, double-precision, floating-point `mxArray`
`mxCreateDoubleScalar`	Create scalar, double-precision array initialized to the specified value
`mxCreateLogicalArray`	Create N-dimensional, logical `mxArray` initialized to `false`
`mxCreateLogicalMatrix`	Create two-dimensional, logical `mxArray` initialized to `false`
`mxCreateLogicalScalar`	Create scalar, logical `mxArray` initialized to `false`
`mxCreateFull (Obsolete)`	Use `mxCreateDoubleMatrix`
`mxCreateNumericArray`	Create unpopulated N-dimensional numeric `mxArray`
`mxCreateNumericMatrix`	Create numeric matrix and initialize data elements to 0
`mxCreateScalarDouble`	Create scalar, double-precision array initialized to specified value
`mxCreateSparse`	Create two-dimensional unpopulated sparse `mxArray`
`mxCreateSparseLogicalMatrix`	Create unpopulated, two-dimensional, sparse, logical `mxArray`
`mxCreateString`	Create 1-by-n string `mxArray` initialized to specified string
`mxCreateStructArray`	Create unpopulated N-dimensional structure `mxArray`
`mxCreateStructMatrix`	Create unpopulated two-dimensional structure `mxArray`
`mxDestroyArray`	Free dynamic memory allocated by an `mxCreate` routine

`mxDuplicateArray`	Make deep copy of array
`mxFree`	Free dynamic memory allocated by `mxCalloc`
`mxFreeMatrix (Obsolete)`	Use `mxDestroyArray`
`mxGetCell`	Get cell's contents
`mxGetChars`	Get pointer to character array data
`mxGetClassID`	Get `mxArray s` class
`mxGetClassName`	Get `mxArray s` class
`mxGetData`	Get pointer to data
`mxGetDimensions`	Get pointer to dimensions array
`mxGetElementSize`	Get number of bytes required to store each data element
`mxGetEps`	Get value of `eps`
`mxGetField`	Get field value, given field name and index in structure array
`mxGetFieldByNumber`	Get field value, given field number and index in structure array
`mxGetFieldNameByNumber`	Get field name, given field number in structure array
`mxGetFieldNumber`	Get field number, given field name in structure array
`mxGetImagData`	Get pointer to imaginary data of `mxArray`
`mxGetInf`	Get value of infinity
`mxGetIr`	Get `ir` array of sparse matrix
`mxGetJc`	Get `jc` array of sparse matrix
`mxGetLogicals`	Get pointer to logical array data
`mxGetM`	Get number of rows

`mxGetN`	Get number of columns or number of elements
`mxGetName (Obsolete)`	Get name of specified `mxArray`
`mxGetNaN`	Get the value of `NaN`
`mxGetNumberOfDimensions`	Get number of dimensions
`mxGetNumberOfElements`	Get number of elements in array
`mxGetNumberOfFields`	Get number of fields in structure `mxArray`
`mxGetNzmax`	Get number of elements in `ir`, `pr`, and `pi` arrays
`mxGetPi`	Get `mxArray`'s imaginary data elements
`mxGetPr`	Get `mxArray`'s real data elements
`mxGetScalar`	Get real component of `mxArray s` first data element
`mxGetString`	Copy string `mxArray s` data into C-style string
`mxIsCell`	True if cell `mxArray`
`mxIsChar`	True if string `mxArray`
`mxIsClass`	True if `mxArray` is member of specified class
`mxIsComplex`	True if data is complex
`mxIsDouble`	True if `mxArray` represents its data as double-precision, floating-point numbers
`mxIsEmpty`	True if `mxArray` is empty
`mxIsFinite`	True if value is finite
`mxIsFromGlobalWS`	True if `mxArray` was copied from the MATLAB global workspace

`mxIsFull (Obsolete)`	Use `mxIsSparse`
`mxIsInf`	True if value is infinite
`mxIsInt8`	True if `mxArray` represents its data as signed 8-bit integers
`mxIsInt16`	True if `mxArray` represents its data as signed 16-bit integers
`mxIsInt32`	True if `mxArray` represents its data as signed 32-bit integers
`mxIsLogical`	True if `mxArray` is Boolean
`mxIsLogicalScalar`	True if scalar `mxArray` of class mxLOGICAL
`mxIsLogicalScalarTrue`	True if scalar `mxArray` of class mxLOGICAL is true
`mxIsNaN`	True if value is `NaN`
`mxIsNumeric`	True if `mxArray` is numeric
`mxIsSingle`	True if `mxArray` represents its data as single-precision, floating-point numbers
`mxIsSparse`	True if sparse `mxArray`
`mxIsString (Obsolete)`	Use `mxIsChar`
`mxIsStruct`	True if structure `mxArray`
`mxIsUint8`	True if `mxArray` represents its data as unsigned 8-bit integers
`mxIsUint16`	True if `mxArray` represents its data as unsigned 16-bit integers
`mxIsUint32`	True if `mxArray` represents its data as unsigned 32-bit integers
`mxMalloc`	Allocate dynamic memory using the MATLAB memory manager
`mxRealloc`	Reallocate memory

`mxRemoveField`	Remove field from structure array
`mxSetAllocFcns`	Register memory allocation/ deallocation functions in stand-alone engine or MAT application
`mxSetCell`	Set value of one cell
`mxSetClassName`	Convert MATLAB structure array to MATLAB object array
`mxSetData`	Set pointer to data
`mxSetDimensions`	Modify number/size of dimensions
`mxSetField`	Set field value of structure array, given field name/index
`mxSetFieldByNumber`	Set field value in structure array, given field number/index
`mxSetImagData`	Set imaginary data pointer for `mxArray`
`mxSetIr`	Set `ir` array of sparse `mxArray`
`mxSetJc`	Set `jc` array of sparse `mxArray`
`mxSetLogical (Obsolete)`	Set logical flag
`mxSetM`	Set number of rows
`mxSetN`	Set number of columns
`mxSetName (Obsolete)`	Set name of `mxArray`
`mxSetNzmax`	Set storage space for nonzero elements
`mxSetPi`	Set new imaginary data for `mxArray`
`mxSetPr`	Set new real data for `mxArray`

Purpose

Add a field to a structure array

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

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

Returns

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

Description

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

See Also

`mxRemoveField`, `mxSetFieldByNumber`

mxArrayToString

Purpose

Convert arrays to strings

C Syntax

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

Arguments

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

Returns

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

Description

Call `mxArrayToString` to copy the character data of a string `mxArray` into a C-style string. The C-style string is always terminated with a `NULL` character.

If the string array contains several rows, they are copied, one column at a time, into one long string array. This function is similar to `mxGetString`, except that:

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

`mxArrayToString` does not free the dynamic memory that the `char` pointer points to. Consequently, you should typically free the string (using `mxFree`) immediately after you have finished using it.

Examples

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

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

See Also

`mxCreateCharArray`, `mxCreateCharMatrixFromStrings`, `mxCreateString`, `mxGetString`

Purpose

Check assertion value for debugging purposes

C Syntax

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

Arguments

`expr`
Value of assertion.

`error_message`
Description of why assertion failed.

Description

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

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

Note that the MEX script turns off these assertions when building optimized MEX-functions, so you should use this for debugging purposes only. 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.

mxAssertS

Purpose

Check assertion value for debugging purposes; doesn't print assertion's text

C Syntax

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

Arguments

`expr`
Value of assertion.

`error_message`
Description of why assertion failed.

Description

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

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

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

Purpose

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

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

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

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

Returns

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

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

Description

Call `mxCalcSingleSubscript` to determine how many elements there are between the beginning of the `mxArray` and a given element of that `mxArray`. For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the `(0,0)` element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have.

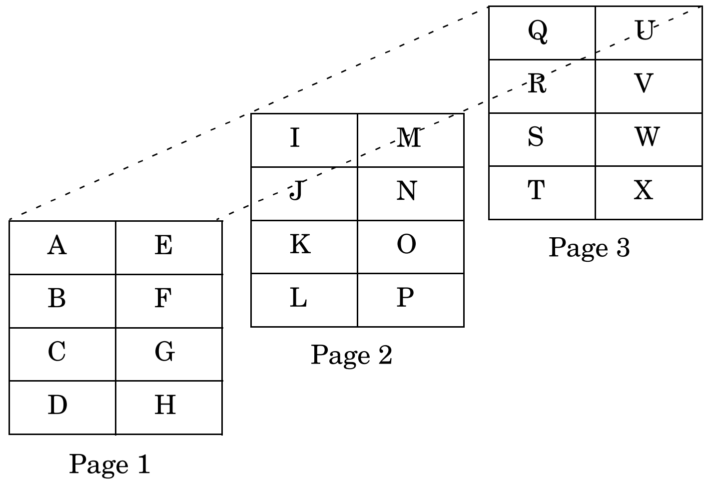
MATLAB uses a column-major numbering scheme to represent data elements internally. That means that MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on through the last column. For example, suppose you create a 4-by-2 variable.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

A	E
B	F
C	G
D	H

Although in fact, MATLAB internally represents the data as the following:

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

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



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

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

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

Examples

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

mxCalloc

Purpose

Allocate dynamic memory using the MATLAB memory manager

C Syntax

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

Arguments

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

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

Returns

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

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

Description

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

In MEX-files, `mxCalloc` automatically

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

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

In stand-alone MATLAB applications, `mxCalloc` defaults to calling the ANSI C `calloc` function. If this default behavior is unacceptable, you can write your own memory allocation routine, and then register this routine with `mxSetAllocFcns`. Then, whenever `mxCalloc` is called, `mxCalloc` calls your memory allocation routine instead of `calloc`.

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

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

Examples

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

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

See Also

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

mxChar

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

C Syntax

```
typedef Uint16 mxChar;
```

Description All string mxArrays store their data elements as mxChar rather than as char. The MATLAB API defines an mxChar as a 16-bit unsigned integer.

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

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

See Also mxCreateCharArray

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

C Syntax

```
typedef enum {
        mxUNKNOWN_CLASS = 0,
        mxCELL_CLASS,
        mxSTRUCT_CLASS,
        mxOBJECT_CLASS,
        mxCHAR_CLASS,
        mxLOGICAL_CLASS,
        mxDOUBLE_CLASS,
        mxSINGLE_CLASS,
        mxINT8_CLASS,
        mxUINT8_CLASS,
        mxINT16_CLASS,
        mxUINT16_CLASS,
        mxINT32_CLASS,
        mxUINT32_CLASS,
        mxINT64_CLASS, /* place holder - future enhancements */
        mxUINT64_CLASS, /* place holder - future enhancements */
       mxFUNCTION_CLASS
} mxClassID;
```

Constants

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

mxCELL_CLASS
Identifies a cell mxArray.

mxSTRUCT_CLASS
Identifies a structure mxArray.

mxOBJECT_CLASS
Identifies a user-defined (nonstandard) mxArray.

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

`mxLOGICAL_CLASS`
Identifies a logical `mxArray`; that is, an `mxArray` that stores Boolean elements, `true` and `false`.

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

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

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

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

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

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

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

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

`mxINT64_CLASS`
Reserved for possible future use.

`mxUINT64_CLASS`
Reserved for possible future use.

`mxFUNCTION_CLASS`
Identifies a function handle `mxArray`.

Description

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

Examples

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

See Also

`mxCreateNumericArray`

Purpose Clear the logical flag

Note As of MATLAB version 6.5, `mxClearLogical` is obsolete. Support for `mxClearLogical` may be removed in a future version.

C Syntax

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

Arguments

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

Description

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

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

Examples

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

See Also

`mxIsLogical`

mxComplexity

Purpose Flag that specifies whether an `mxArray` has imaginary components

C Syntax

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

Constants

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

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

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

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

See Also `mxCreateNumericArray`, `mxCreateDoubleMatrix`, `mxCreateSparse`

Purpose Create unpopulated N-dimensional cell mxArray

C Syntax

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

Arguments

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

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

Returns A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellArray returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. The most common cause of failure is insufficient free heap space.

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

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

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

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

See Also mxCreateCellMatrix, mxGetCell, mxSetCell, mxIsCell

mxCreateCellMatrix

Purpose

Create unpopulated two-dimensional cell `mxArray`

C Syntax

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

Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

Returns

A pointer to the created cell `mxArray`, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateCellMatrix` returns `NULL`. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for `mxCreateCellMatrix` to be unsuccessful.

Description

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

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

Examples

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

See Also

`mxCreateCellArray`

Purpose Create unpopulated N-dimensional string mxArray

C Syntax

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

Arguments

ndim
The desired number of dimensions in the string mxArray. You must specify a positive number. If you specify 0, 1, or 2, mxCreateCharArray creates a two-dimensional mxArray.

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

Returns A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCharArray returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for mxCreateCharArray to be unsuccessful.

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

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

See Also mxCreateCharMatrixFromStrings, mxCreateString

mxCreateCharMatrixFromStrings

Purpose

Create populated two-dimensional string `mxArray`

C Syntax

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

Arguments

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

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

Returns

A pointer to the created string `mxArray`, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateCharMatrixFromStrings` returns `NULL`. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the primary reason for `mxCreateCharArray` to be unsuccessful. Another possible reason for failure is that `str` contains fewer than `m` strings.

Description

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

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

Examples

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

See Also

`mxCreateCharArray`, `mxCreateString`, `mxGetString`

Purpose Create unpopulated two-dimensional, double-precision, floating-point mxArray

C Syntax

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

Arguments

m
The desired number of rows.

n
The desired number of columns.

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

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

Description Use mxCreateDoubleMatrix to create an m-by-n mxArray. mxCreateDoubleMatrix initializes each element in the pr array to 0. If you set ComplexFlag to mxCOMPLEX, mxCreateDoubleMatrix also initializes each element in the pi array to 0.

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

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

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

See Also mxCreateNumericArray, mxComplexity

mxCreateDoubleScalar

Purpose

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

---

Note This function replaces `mxCreateScalarDouble` in version 6.5 of MATLAB. `mxCreateScalarDouble` is still supported in version 6.5, but may be removed in a future version.

C Syntax

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

Arguments

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

Returns

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

Description

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

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

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

See Also

`mxGetPr`, `mxCreateDoubleMatrix`

V4 Compatible

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

Use

```
mxCreateDoubleMatrix
```

instead of

```
mxCreateFull
```

See Also

`mxCreateDoubleMatrix`

mxCreateLogicalArray

Purpose

Create N-dimensional logical `mxArray` initialized to `false`

C Syntax

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

Arguments

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

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

Returns

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

Description

Call `mxCreateLogicalArray` to create an N-dimensional `mxArray` of logical (`true` and `false`) elements. After creating the `mxArray`, `mxCreateLogicalArray` initializes all its elements to `false`. `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.

See Also

`mxCreateLogicalMatrix`, `mxCreateSparseLogicalMatrix`, `mxCreateLogicalScalar`

Purpose

Create two-dimensional, logical `mxArray` initialized to `false`

C Syntax

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

Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

Returns

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

Description

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

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

See Also

`mxCreateLogicalArray`, `mxCreateSparseLogicalMatrix`, `mxCreateLogicalScalar`

mxCreateLogicalScalar

Purpose

Create scalar, logical `mxArray` initialized to `false`

C Syntax

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

Arguments

`value`
The desired logical value (`true` or `false`) to which you want to initialize the array.

Returns

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

Description

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

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

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

See Also

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

Purpose Create unpopulated N-dimensional numeric `mxArray`

C Syntax

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

Arguments

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

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

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

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

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

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

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

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

Examples

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

See Also

`mxClassID`, `mxCreateDoubleMatrix`, `mxCreateSparse`, `mxCreateString`, `mxComplexity`

Purpose

Create numeric matrix and initialize all its data elements to 0

C Syntax

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

Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

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

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

Returns

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

Description

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

See Also

`mxCreateNumericArray`

mxCreateScalarDouble

Purpose

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

---

Note This function is replaced by `mxCreateDoubleScalar` in version 6.5 of MATLAB. `mxCreateScalarDouble` is still supported in version 6.5, but may be removed in a future version.

C Syntax

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

Arguments

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

Returns

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

Description

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

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

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

See Also

`mxGetPr`, `mxCreateDoubleMatrix`

Purpose

Create two-dimensional unpopulated sparse mxArray

C Syntax

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

Arguments

m
The desired number of rows.

n
The desired number of columns.

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

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

Returns

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

Description

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

mxCreateSparse allocates space for:

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

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

Examples

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

See Also

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

Purpose

Create unpopulated two-dimensional, sparse, logical mxArray

C Syntax

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

Arguments

m
The desired number of rows.

n
The desired number of columns.

Returns

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

Description

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

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

See Also

mxCreateLogicalMatrix, mxCreateLogicalArray, mxCreateLogicalScalar, mxCreateSparse, mxIsLogical

mxCreateString

Purpose

Create 1-by-n string `mxArray` initialized to the specified string

C Syntax

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

Arguments

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

Returns

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

Description

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

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

Examples

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

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

See Also

`mxCreateCharMatrixFromStrings`, `mxCreateCharArray`

Purpose

Create unpopulated N-dimensional structure `mxArray`

C Syntax

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

Arguments

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

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

`nfields`
The desired number of fields in each element.

`field_names`
The desired list of field names.

Returns

A pointer to the created structure `mxArray` if successful, and `NULL` otherwise. The most likely cause of failure is insufficient heap space to hold the returned `mxArray`.

Description

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

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

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

Examples

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

See Also

`mxDestroyArray`, `mxSetNzmax`

Purpose Create unpopulated two-dimensional structure mxArray

C Syntax

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

Arguments

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

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

nfields
The desired number of fields in each element.

field_names
The desired list of field names.

Returns A pointer to the created structure mxArray if successful, and NULL otherwise. The most likely cause of failure is insufficient heap space to hold the returned mxArray.

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

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

See Also mxCreateStructArray, mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber, mxIsStruct

mxDestroyArray

Purpose Free dynamic memory allocated by an `mxCreate` routine

C Syntax

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

Arguments

`array_ptr`
Pointer to the `mxArray` that you want to free.

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

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

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

See Also `mxCalloc`, `mxFree`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

Purpose

Make a deep copy of an array

C Syntax

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

Arguments

`in`
Pointer to the `mxArray` that you want to copy.

Returns

Pointer to a copy of the array.

Description

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

Examples

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

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

mxFree

Purpose Free dynamic memory allocated by `mxCalloc`

C Syntax

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

Arguments

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

Description

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

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

In MEX-files, `mxFree` automatically

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

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

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

In a MEX-file, your use of `mxFree` depends on whether the specified memory parcel is persistent or nonpersistent. By default, memory parcels created by `mxCalloc` are nonpersistent. However, if an application calls `mexMakeMemoryPersistent`, then the specified memory parcel becomes persistent.

The MATLAB memory management facility automatically frees all nonpersistent memory whenever a MEX-file completes. Thus, even if you do not call `mxFree`, MATLAB takes care of freeing the memory for you. Nevertheless, it is a good programming practice to deallocate memory just as soon as you are through using it. Doing so generally makes the entire system run more efficiently.

When a MEX-file completes, the MATLAB memory management facility does not free persistent memory parcels. Therefore, the only way to free a persistent memory parcel is to call `mxFree`. Typically, MEX-files call `mexAtExit` to register a clean-up handler. Then, the clean-up handler calls `mxFree`.

Examples

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

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

See Also

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

mxFreeMatrix (Obsolete)

V4 Compatible

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

Use

```
mxDestroyArray
```

instead of

```
mxFreeMatrix
```

See Also

`mxDestroyArray`

Purpose

Get a cell's contents

C Syntax

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

Arguments

`array_ptr`
Pointer to a cell `mxArray`.

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

Returns

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

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

Description

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

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

Examples

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

See Also

`mxCreateCellArray`, `mxIsCell`, `mxSetCell`

mxGetChars

Purpose

Get pointer to character array data

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

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

Description

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

See Also

`mxGetString`, `mxGetPr`, `mxGetPi`, `mxGetCell`, `mxGetField`, `mxGetLogicals`, `mxGetScalar`

Purpose

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

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

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

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

`mxCELL_CLASS`
Identifies a cell `mxArray`.

`mxSTRUCT_CLASS`
Identifies a structure `mxArray`.

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

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

`mxLOGICAL_CLASS`
Identifies a logical `mxArray`; that is, an `mxArray` that stores logical values representing `true` and `false`.

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

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

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

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

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

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

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

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

mxINT64_CLASS
Reserved for possible future use.

mxUINT64_CLASS
Reserved for possible future use.

mxFUNCTION_CLASS
Identifies a function handle mxArray.

Description

Use mxGetClassId to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For example, if array_ptr points to a logical mxArray, then mxGetClassID returns mxLOGICAL_CLASS.

mxGetClassID is similar to mxGetClassName, except that the former returns the class as an enumerated value and the latter returns the class as a string.

Examples

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

See Also

mxGetClassName

Purpose Get (as a string) an `mxArray` s class

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

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

Description Call `mxGetClassName` to determine the class of an `mxArray`. The class of an `mxArray` identifies the kind of data the `mxArray` is holding. For example, if `array_ptr` points to a sparse `mxArray`, then `mxGetClassName` returns `sparse`.

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

Examples See `mexfunction.c` in the `mex` subdirectory of the `examples` directory. For an additional example, see `mxisclass.c` in the `mx` subdirectory of the `examples` directory.

See Also `mxGetClassID`

mxGetData

Purpose Get pointer to data

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Description Similar to mxGetPr, except mxGetData returns a void *. Use mxGetData on numeric arrays with contents other than double.

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

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

See Also mxGetPr

Purpose

Get a pointer to the dimensions array

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

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

Description

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

Examples

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

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

See Also

`mxGetNumberOfDimensions`

mxGetElementSize

Purpose Get the number of bytes required to store each data element

C Syntax

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

Arguments `array_ptr`
Pointer to an `mxArray`.

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

Description Call `mxGetElementSize` to determine the number of bytes in each data element of the `mxArray`. For example, if the `mxClassID` of an `mxArray` is `mxINT16_CLASS`, then the `mxArray` stores each data element as a 16-bit (2 byte) signed integer. Thus, `mxGetElementSize` returns 2.

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

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

See Also `mxGetM`, `mxGetN`

Purpose

Get value of `eps`

C Syntax

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

Returns

The value of the MATLAB `eps` variable.

Description

Call `mxGetEps` to return the value of the MATLAB `eps` variable. This variable holds the distance from 1.0 to the next largest floating-point number. As such, it is a measure of floating-point accuracy. The MATLAB `PINV` and `RANK` functions use `eps` as a default tolerance.

Examples

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

See Also

`mxGetInf`, `mxGetNaN`

mxGetField

Purpose

Get a field value, given a field name and an index in a structure array

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

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

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

Returns

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

- Specifying an `array_ptr` that does not point to a structure `mxArray`. To determine if `array_ptr` points to a structure `mxArray`, call `mxIsStruct`.
- Specifying an out-of-range `index` to an element past the end of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than 9.
- Specifying a nonexistent `field_name`. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to get existing `field names`.
- Insufficient heap space to hold the returned `mxArray`.

Description

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

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

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

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

Calling

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

is equivalent to calling

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

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

See Also

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

mxGetFieldByNumber

Purpose Get a field value, given a field number and an index in a structure array

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

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

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

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

- Specifying an `array_ptr` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to is a structure `mxArray`.
- Specifying an index < 0 or >= the number of elements in the array.
- Specifying a nonexistent field number. Call `mxGetFieldNumber` to determine the field number that corresponds to a given field name.

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

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

Calling

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

is equivalent to calling

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

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

Examples

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

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

See Also

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

mxGetFieldNameByNumber

Purpose Get a field name, given a field number in a structure array

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

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

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

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

Description Call `mxGetFieldNameByNumber` to get the name of a field in the given structure `mxArray`. A typical use of `mxGetFieldNameByNumber` is to call it inside a loop in order to get the names of all the fields in a given `mxArray`.

Consider a MATLAB structure initialized to

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

The `field_number` 0 represents the field name `name`; `field_number` 1 represents field name `billing`; `field_number` 2 represents field name `test`. A `field_number` other than 0, 1, or 2 causes `mxGetFieldNameByNumber` to return `NULL`.

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

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

See Also `mxGetField`, `mxIsStruct`, `mxSetField`

mxGetFieldNumber

Purpose

Get a field number, given a field name in a structure array

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

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

Returns

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

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

Description

If you know the name of a field but do not know its field number, call `mxGetFieldNumber`. Conversely, if you know the field number but do not know its field name, call `mxGetFieldNameByNumber`.

For example, consider a MATLAB structure initialized to

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

The `field_name` `"name"` has a field number of 0; the `field_name` `"billing"` has a `field_number` of 1; and the `field_name` `"test"` has a field number of 2. If you call `mxGetFieldNumber` and specify a `field_name` of anything other than `"name"`, `"billing"`, or `"test"`, then `mxGetFieldNumber` returns `-1`.

Calling

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

is equivalent to calling

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

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

Examples

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

See Also

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

mxGetImagData

Purpose Get pointer to imaginary data of an mxArray

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Description Similar to mxGetPi, except it returns a void *. Use mxGetImagData on numeric arrays with contents other than double.

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

See Also mxGetPi

Purpose Get the value of infinity

C Syntax

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

Returns The value of infinity on your system.

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

Operations that return infinity include:

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

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

See Also `mxGetEps`, `mxGetNaN`

mxGetIr

Purpose

Get the ir array of a sparse matrix

C Syntax

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

Arguments

array_ptr
Pointer to a sparse mxArray.

Returns

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

- Specifying a full (nonsparse) mxArray.
- Specifying a NULL array_ptr. (This usually means that an earlier call to mxCreateSparse failed.)

Description

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

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

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

Examples

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

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

See Also

mxGetJc, mxGetNzmax, mxSetIr, mxSetJc, mxSetNzmax

Purpose

Get the jc array of a sparse matrix

C Syntax

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

Arguments

array_ptr
Pointer to a sparse mxArray.

Returns

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

Description

Use mxGetJc to obtain the starting address of the jc array. The jc array is an integer array having n+1 elements where n is the number of columns in the sparse mxArray. The values in the jc array indirectly indicate columns containing nonzero elements. For a detailed explanation of the jc array, see mxSetJc.

Examples

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

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

See Also

mxGetIr, mxSetIr, mxSetJc

mxGetLogicals

Purpose Get pointer to logical array data

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

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

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

See Also `mxIsLogical`, `mxIsLogicalScalar`, `mxIsLogicalScalarTrue`, `mxCreateLogicalScalar`, `mxCreateLogicalMatrix`, `mxCreateLogicalArray`

Purpose Get the number of rows

C Syntax

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

Arguments

`array_ptr`
Pointer to an array.

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

Description `mxGetM` returns the number of rows in the specified array. The term *rows* always means the first dimension of the array no matter how many dimensions the array has. For example, if `array_ptr` points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, then `mxGetM` returns 8.

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

For additional examples, see `fulltosparse.c`, `revord.c`, `timestwo.c`, and `xtimesy.c` in the `refbook` subdirectory of the `examples` directory; see `mxmalloc.c` and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory; see `mexget.c`, `mexlock.c`, `mexsettrapflag.c`, and `yprime.c` in the `mex` subdirectory of the `examples` directory.

See Also `mxGetN`, `mxSetM`, `mxSetN`

mxGetN

Purpose

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

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

The number of columns in the `mxArray`.

Description

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

If `array_ptr` is an N-dimensional `mxArray`, `mxGetN` is the product of dimensions 2 through N. For example, if `array_ptr` points to a four-dimensional `mxArray` having dimensions 13-by-5-by-4-by-6, then `mxGetN` returns the value 120 (5x4x6). If the specified `mxArray` has more than two dimensions and you need to know exactly how many elements are in each dimension, then call `mxGetDimensions`.

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

Examples

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

For additional examples,

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

See Also

`mxGetM`, `mxGetNumberOfDimensions`, `mxSetM`, `mxSetN`

V5 Compatible

This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

mxGetNaN

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

C Syntax

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

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

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

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

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

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

See Also `mxGetEps`, `mxGetInf`

Purpose Get the number of dimensions

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns The number of dimensions in the specified `mxArray`. The returned value is always 2 or greater.

Description Use `mxGetNumberOfDimensions` to determine how many dimensions are in the specified array. To determine how many elements are in each dimension, call `mxGetDimensions`.

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

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

See Also `mxSetM`, `mxSetN`, `mxGetDimensions`

mxGetNumberOfElements

Purpose Get number of elements in an array

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns Number of elements in the specified `mxArray`.

Description `mxGetNumberOfElements` tells you how many elements an array has. For example, if the dimensions of an array are 3-by-5-by-10, then `mxGetNumberOfElements` will return the number 150.

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

For additional examples, see `explore.c` in the `mex` subdirectory of the `examples` directory; see `mxcalcsinglesubscript.c`, `mxgeteps.c`, `mxgetinf.c`, `mxisfinite.c`, and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

See Also `mxGetDimensions`, `mxGetM`, `mxGetN`, `mxGetClassID`, `mxGetClassName`

Purpose

Get the number of fields in a structure `mxArray`

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

Returns

The number of fields, on success. Returns 0 on failure. The most common cause of failure is that `array_ptr` is not a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` is a structure.

Description

Call `mxGetNumberOfFields` to determine how many fields are in the specified structure `mxArray`.

Once you know the number of fields in a structure, it is easy to loop through every field in order to set or to get field values.

Examples

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

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

See Also

`mxGetField`, `mxIsStruct`, `mxSetField`

mxGetNzmax

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

C Syntax

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

Arguments

`array_ptr`
Pointer to a sparse `mxArray`.

Returns The number of elements allocated to hold nonzero entries in the specified sparse `mxArray`, on success. Returns an indeterminate value on error. The most likely cause of failure is that `array_ptr` points to a full (nonsparse) `mxArray`.

Description Use `mxGetNzmax` to get the value of the `nzmax` field. The `nzmax` field holds an integer value that signifies the number of elements in the `ir`, `pr`, and, if it exists, the `pi` arrays. The value of `nzmax` is always greater than or equal to the number of nonzero elements in a sparse `mxArray`. In addition, the value of `nzmax` is always less than or equal to the number of rows times the number of columns.

As you adjust the number of nonzero elements in a sparse `mxArray`, MATLAB often adjusts the value of the `nzmax` field. MATLAB adjusts `nzmax` in order to reduce the number of costly reallocations and in order to optimize its use of heap space.

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

See Also `mxSetNzmax`

Purpose Get an mxArray's imaginary data elements

C Syntax

```
#include "matrix.h"
double *mxGetPi(const mxArray *array_ptr);
```

Arguments

array_ptr
Pointer to an mxArray.

Returns The imaginary data elements of the specified mxArray, on success. Returns NULL if there is no imaginary data or if there is an error.

Description The pi field points to an array containing the imaginary data of the mxArray. Call mxGetPi to get the contents of the pi field; that is, to get the starting address of this imaginary data.

The best way to determine if an mxArray is purely real is to call mxIsComplex.

The imaginary parts of all input matrices to a MATLAB function are allocated if any of the input matrices are complex.

Examples See convec.c, findnz.c, and fulltosparse.c in the refbook subdirectory of the examples directory.

For additional examples, see explore.c and mexcallmatlab.c in the mex subdirectory of the examples directory; see mxcalcsinglesubscript.c, mxgetinf.c, mxisfinite.c, and mxsetnzmax.c in the mx subdirectory of the examples directory.

See Also mxGetPr, mxSetPi, mxSetPr

mxGetPr

Purpose

Get an mxArray's real data elements

C Syntax

```
#include "matrix.h"
double *mxGetPr(const mxArray *array_ptr);
```

Arguments

array_ptr
Pointer to an mxArray.

Returns

The address of the first element of the real data. Returns NULL if there is no real data.

Description

Call mxGetPr to determine the starting address of the real data in the mxArray that array_ptr points to. Once you have the starting address, you can access any other element in the mxArray.

Examples

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

See Also

mxGetPi, mxSetPi, mxSetPr

Purpose

Get the real component of an mxArray s first data element

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray other than a cell mxArray or a structure mxArray.

Returns

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

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

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

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

Description

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

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

Examples

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

For additional examples, see mxsetdimensions.c in the mx subdirectory of the examples directory; see mexget.c, mexlock.c and mexsettrapflag.c in the mex subdirectory of the examples directory.

See Also `mxGetM`, `mxGetN`

Purpose

Copy a string mxArray s data into a C-style string

C Syntax

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

Arguments

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

buf
The starting location into which the string should be written. mxGetString writes the character data into buf and then terminates the string with a NULL character (in the manner of C strings). buf can either point to dynamic or static memory.

buflen
Maximum number of characters to read into buf. Typically, you set buflen to 1 plus the number of elements in the string mxArray to which array_ptr points. See the mxGetM and mxGetN reference pages to find out how to get the number of elements.

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

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

Returns

0 on success, and 1 on failure. Possible reasons for failure include:

- Specifying an mxArray that is not a string mxArray.
- Specifying buflen with less than the number of characters needed to store the entire mxArray pointed to by array_ptr. If this is the case, 1 is returned and the string is truncated.

Description

Call mxGetString to copy the character data of a string mxArray into a C-style string. The copied C-style string starts at buf and contains no more than buflen-1 characters. The C-style string is always terminated with a NULL character.

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

Examples

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

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

See Also

mxCreateCharArray, mxCreateCharMatrixFromStrings, mxCreateString

Purpose

True if a cell `mxArray`

C Syntax

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

Arguments

`array_ptr`
Pointer to an array.

Returns

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

Description

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

Calling `mxIsCell` is equivalent to calling

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

Note `mxIsCell` does not answer the question, “Is this `mxArray` a cell of a cell array?”. An individual cell of a cell array can be of any type.

See Also

`mxIsClass`

mxIsChar

Purpose True if a string mxArray

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

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

Description Use mxIsChar to determine if array_ptr points to string mxArray.

Calling mxIsChar is equivalent to calling

```
mxGetClassID(array_ptr) == mxCHAR_CLASS
```

Examples See phonebook.c and revord.c in the refbook subdirectory of the examples directory.

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

See Also mxIsClass, mxGetClassID

Purpose

True if mxArray is a member of the specified class

C Syntax

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

Arguments

array_ptr
Pointer to an array.

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

Value of Name	Corresponding Class
cell	mxCELL_CLASS
char	mxCHAR_CLASS
double	mxDOUBLE_CLASS
function handle	mxFUNCTION_CLASS
int8	mxINT8_CLASS
int16	mxINT16_CLASS
int32	mxINT32_CLASS
logical	mxLOGICAL_CLASS
single	mxSINGLE_CLASS
struct	mxSTRUCT_CLASS
uint8	mxUINT8_CLASS
uint16	mxUINT16_CLASS
uint32	mxUINT32_CLASS
<class_name>	mxOBJECT_CLASS
unknown	mxUNKNOWN_CLASS

In the table, <*class_name*> represents the name of a specific MATLAB custom object.

Or, you can specify one of your own class names.

For example,

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

is equivalent to calling

```
mxIsDouble(array_ptr);
```

which is equivalent to calling

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

Note that it is most efficient to use the `mxIsDouble` form.

Returns

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

Description

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

Examples

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

See Also

`mxIsEmpty`, `mxGetClassID`, `mxClassID`

Purpose

True if data is complex

C Syntax

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

Returns

`true` if `array_ptr` is a numeric array containing complex data, and `false` otherwise. If `array_ptr` points to a cell array or a structure array, then `mxIsComplex` returns `false`.

Description

Use `mxIsComplex` to determine whether or not an imaginary part is allocated for an `mxArray`. The imaginary pointer `pi` is `NULL` if an `mxArray` is purely real and does not have any imaginary data. If an `mxArray` is complex, `pi` points to an array of numbers.

Examples

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

For additional examples, see `convec.c`, `phonebook.c`, `timestwo.c`, and `xtimesy.c` in the `refbook` subdirectory of the `examples` directory; see `explore.c`, `yprime.c`, `mexlock.c`, and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory; see `mxcalcsinglesubscript.c`, `mxgeteps.c`, and `mxgetinf.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxIsNumeric`

mxIsDouble

Purpose

True if mxArray represents its data as double-precision, floating-point numbers

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

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

Description

Call mxIsDouble to determine whether or not the specified mxArray represents its real and imaginary data as double-precision, floating-point numbers.

Older versions of MATLAB store all mxArray data as double-precision, floating-point numbers. However, starting with MATLAB version 5, MATLAB can store real and imaginary data in a variety of numerical formats.

Calling mxIsDouble is equivalent to calling

```
mxGetClassID(array_ptr == mxDOUBLE_CLASS)
```

Examples

See findnz.c, fulltosparse.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory.

For additional examples, see mexget.c, mexlock.c, mexsettrapflag.c, and yprime.c in the mex subdirectory of the examples directory; see mxcalcsinglesubscript.c, mxgeteps.c, mxgetinf.c, and mxisfinite.c in the mx subdirectory of the examples directory.

See Also

mxIsClass, mxGetClassID

Purpose

True if `mxArray` is empty

C Syntax

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

Arguments

`array_ptr`
Pointer to an array.

Returns

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

Description

Use `mxIsEmpty` to determine if an `mxArray` contains no data. An `mxArray` is empty if the size of any of its dimensions is 0.

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

Examples

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

See Also

`mxIsClass`

mxIsFinite

Purpose True if value is finite

C Syntax

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

Arguments

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

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

Description Call `mxIsFinite` to determine whether or not `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

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

See Also `mxIsInf`, `mxIsNaN`

Purpose True if the mxArray was copied from the MATLAB global workspace

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

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

Description mxIsFromGlobalWS is useful for stand-alone MAT programs. mexIsGlobal tells you if the pointer you pass actually points into the global workspace.

Examples See matdgns.c and matcreat.c in the eng_mat subdirectory of the examples directory.

See Also mexIsGlobal

mxIsFull (Obsolete)

V4 Compatible

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

Use

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

instead of

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

See Also

`mxIsSparse`

Purpose True if value is infinite

C Syntax

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

Arguments

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

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

Description Call `mxIsInf` to determine whether or not `value` is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable, `Inf`, is built into the system; you cannot modify it.

Operations that return infinity include:

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

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

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

See Also `mxIsFinite`, `mxIsNaN`

mxIsInt8

Purpose True if mxArray represents its data as signed 8-bit integers

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns true if the array stores its data as signed 8-bit integers, and false otherwise.

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

Calling mxIsInt8 is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT8_CLASS
```

See Also mxIsClass, mxGetClassID

Purpose True if mxArray represents its data as signed 16-bit integers

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns true if the array stores its data as signed 16-bit integers, and false otherwise.

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

Calling mxIsInt16 is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT16_CLASS
```

See Also mxIsClass, mxGetClassID

mxIsInt32

Purpose

True if `mxArray` represents its data as signed 32-bit integers

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

`true` if the array stores its data as signed 32-bit integers, and `false` otherwise.

Description

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

Calling `mxIsInt32` is equivalent to calling

```
mxGetClassID(array_ptr) == mxINT32_CLASS
```

See Also

`mxIsClass`, `mxGetClassID`

Purpose True if mxArray is of class mxLOGICAL

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

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

Description

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

If an mxArray is logical, then MATLAB treats all zeros as meaning false and all nonzero values as meaning true. For additional information on the use of logical variables in MATLAB, type help logical at the MATLAB prompt.

Examples

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

See Also

mxIsClass, mxSetLogical (Obsolete)

mxIsLogicalScalar

Purpose True if scalar mxArray of class mxLOGICAL

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

true if the mxArray is of class mxLOGICAL and has 1-by-1 dimensions, and false otherwise.

Description

Use mxIsLogicalScalar to determine whether MATLAB treats the scalar data in the mxArray as logical or numerical. For additional information on the use of logical variables in MATLAB, type help logical at the MATLAB prompt.

mxIsLogicalScalar(pa) is equivalent to

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1
```

See Also

mxIsLogicalScalarTrue, mxIsLogical, mxGetLogicals, mxGetScalar

Purpose True if scalar `mxArray` of class `mxLOGICAL` is `true`

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

`true` if the value of the `mxArray` s logical, scalar element is `true`, and `false` otherwise.

Description

Use `mxIsLogicalScalarTrue` to determine whether the value of a scalar `mxArray` is `true` or `false`. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

`mxIsLogicalScalarTrue(pa)` is equivalent to

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1 &&
mxGetLogicals(pa)[0] == true
```

See Also

`mxIsLogicalScalar`, `mxIsLogical`, `mxGetLogicals`, `mxGetScalar`

mxIsNaN

Purpose True if value is NaN (Not-a-Number)

C Syntax

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

Arguments

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

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

Description Call `mxIsNaN` to determine whether or not `value` is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. A `NaN` is obtained as a result of mathematically undefined operations such as

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

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value, rather it is a family of numbers that MATLAB (and other IEEE-compliant applications) use to represent an error condition or missing data.

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

For additional examples, see `findnz.c` and `fulltosparse.c` in the `refbook` subdirectory of the `examples` directory.

See Also `mxIsFinite`, `mxIsInf`

Purpose True if mxArray is numeric

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

true if the array's storage type is:

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

false if the array's storage type is:

- mxCELL_CLASS
- mxCHAR_CLASS
- mxFUNCTION_CLASS
- mxLOGICAL_CLASS
- mxOBJECT_CLASS
- mxSTRUCT_CLASS
- mxUNKNOWN_CLASS

Description

Call mxIsNumeric to determine if the specified array contains numeric data. If the specified array is a cell, string, or a structure, then mxIsNumeric returns false. Otherwise, mxIsNumeric returns true.

Call mxGetClassID to determine the exact storage type.

Examples

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

See Also

`mxGetClassID`

Purpose

True if mxArray represents its data as single-precision, floating-point numbers

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

Returns

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

Description

Use `mxIsSingle` to determine whether or not the specified array represents its real and imaginary data as single-precision, floating-point numbers.

Calling `mxIsSingle` is equivalent to calling

```
mxGetClassID(array_ptr) == mxSINGLE_CLASS
```

See Also

`mxIsClass`, `mxGetClassID`

mxIsSparse

Purpose True if a sparse mxArray

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

true if array_ptr points to a sparse mxArray, and false otherwise. A false return value means that array_ptr points to a full mxArray or that array_ptr does not point to a legal mxArray.

Description

Use mxIsSparse to determine if array_ptr points to a sparse mxArray. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.

Examples

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

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

See Also

mxGetIr, mxGetJc

V4 Compatible

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

Use

```
mxIsChar
```

instead of

```
mxIsString
```

See Also

`mxChar`, `mxIsChar`

mxIsStruct

Purpose True if a structure `mxArray`

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

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

Description Use `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`. Many routines (for example, `mxGetFieldName` and `mxSetField`) require a structure `mxArray` as an argument.

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

See Also `mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetNumberOfFields`, `mxGetField`, `mxSetField`

Purpose

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

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

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

Description

Use mxIsInt8 to determine whether or not the specified mxArray represents its real and imaginary data as 8-bit unsigned integers.

Calling mxIsUint8 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT8_CLASS
```

See Also

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

mxIsUint16

Purpose

True if mxArray represents its data as unsigned 16-bit integers

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

true if the mxArray stores its data as unsigned 16-bit integers, and false otherwise.

Description

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

Calling mxIsUint16 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT16_CLASS
```

See Also

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

Purpose

True if mxArray represents its data as unsigned 32-bit integers

C Syntax

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

Arguments

array_ptr
Pointer to an mxArray.

Returns

true if the mxArray stores its data as unsigned 32-bit integers, and false otherwise.

Description

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

Calling mxIsUint32 is equivalent to calling

```
mxGetClassID(array_ptr) == mxUINT32_CLASS
```

See Also

mxIsClass, mxGetClassID, mxIsUint16, mxIsUint8, mxIsInt32, mxIsInt16, mxIsInt8

mxMalloc

Purpose Allocate dynamic memory using the MATLAB memory manager

C Syntax

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

Arguments

`n`
Number of bytes to allocate.

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

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

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

In MEX-files, `mxMalloc` automatically

- Allocates enough contiguous heap space to hold `n` bytes.
- Registers the returned heap space with the MATLAB memory management facility.

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

In stand-alone MATLAB applications, `mxMalloc` defaults to calling the ANSI C `malloc` function. If this default behavior is unacceptable, you can write your own memory allocation routine, and then register this routine with `mxSetAllocFcns`. Then, whenever `mxMalloc` is called, `mxMalloc` calls your memory allocation routine instead of `malloc`.

By default, in a MEX-file, `mxMalloc` generates nonpersistent `mxMalloc` data. In other words, the memory management facility automatically deallocates the memory as soon as the MEX-file ends. If you want the memory to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxMalloc`.

If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX-file is cleared.

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

Examples

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

See Also

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

mxRealloc

Purpose Reallocate memory

C Syntax

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

Arguments

`ptr`
Pointer to a block of memory allocated by `mxCalloc`, or by a previous call to `mxRealloc`.

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

Returns A pointer to the reallocated block of memory on success, and `0` on failure.

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

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

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

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

See Also `mxCalloc`, `mxFree`, `mxMalloc`, `mxSetAllocFcns`

Purpose

Remove a field from a structure array

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`.

`field_number`
The number of the field you want to remove. For instance, to remove the first field, set `field_number` to 0; to remove the second field, set `field_number` to 1; and so on.

Description

Call `mxRemoveField` to remove a field from a structure array. If the field does not exist, nothing happens. This function does not destroy the field values. Use `mxDestroyArray` to destroy the actual field values.

Consider a MATLAB structure initialized to

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

The `field_number` 0 represents the field name `name`; `field_number` 1 represents field name `billing`; `field_number` 2 represents field name `test`.

See Also

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

mxSetAllocFcns

Purpose

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

C Syntax

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

Arguments

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

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

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

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

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

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

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

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

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

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

`ptr` Pointer to beginning of the memory parcel to reallocate.

`size` The size of each element. To get the size, you typically use the `sizeof` operator or the `mxGetElementSize` routine.

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

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

`n` The number of bytes to allocate.

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

Description

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

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

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

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

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

Examples

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

See Also

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

mxSetCell

Purpose Set the value of one cell

C Syntax

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

Arguments

`array_ptr`
Pointer to a cell `mxArray`.

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

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

Description

Call `mxSetCell` to put the designated `value` into a particular cell of a cell `mxArray`. You can assign new values to unpopulated cells or overwrite the value of an existing cell. To do the latter, first use `mxDestroyArray` to free what is already there and then `mxSetCell` to assign the new value.

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

Examples

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

See Also

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

Purpose

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

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray` of class `mxSTRUCT_CLASS`.

`classname`
The object class to which to convert `array_ptr`.

Returns

0 if successful, and nonzero otherwise.

Description

`mxSetClassName` converts a structure array to an object array, to be saved subsequently to a MAT-file. The object is not registered or validated by MATLAB until it is loaded via the `LOAD` command. If the specified classname is an undefined class within MATLAB, `LOAD` converts the object back to a simple structure array.

See Also

`mxIsClass`, `mxGetClassID`

mxSetData

Purpose

Set pointer to data

C Syntax

```
#include "matrix.h"
void mxSetData(mxArray *array_ptr, void *data_ptr);
```

Arguments

`array_ptr`
Pointer to an `mxArray`.

`data_ptr`
Pointer to data.

Description

`mxSetData` is similar to `mxSetPr`, except its `data_ptr` argument is a `void *`. Use this on numeric arrays with contents other than `double`.

See Also

`mxSetPr`

mxSetDimensions

Purpose

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

C Syntax

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

Arguments

`array_ptr`
Pointer to an `mxArray`.

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

`ndim`
The desired number of dimensions.

Returns

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

Description

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

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

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

Examples

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

See Also

`mxGetNumberOfDimensions`, `mxSetM`, `mxSetN`

mxSetField

Purpose

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

C Syntax

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

Arguments

`array_ptr`
Pointer to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.

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

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

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

Description

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

```
array_ptr[index].field_name = value;
```

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

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

Calling

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

is equivalent to calling

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

Examples

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

See Also

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

mxSetFieldByNumber

Purpose

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

C Syntax

```
#include "matrix.h"
void mxSetFieldByNumber(mxArray *array_ptr, int index,
     int field_number, mxArray *value);
```

Arguments

`array_ptr`
Pointer to a structure `mxArray`. Call `mxIsStruct` to determine if `array_ptr` points to a structure `mxArray`.

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

`field_number`
The position of the field whose value you want to extract. The first field within each element has a `field_number` of 0, the second field has a `field_number` of 1, and so on. The last field has a `field_number` of `N-1`, where `N` is the number of fields.

`value`
The value you are assigning.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

Description

Use `mxSetFieldByNumber` to assign a `value` to the specified element of the specified field. `mxSetFieldByNumber` is almost identical to `mxSetField`; however, the former takes a field number as its third argument and the latter takes a field name as its third argument.

Calling

```
mxSetField(pa, index, "field_name", new_value_pa);
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`

mxSetImagData

Purpose

Set imaginary data pointer for an `mxArray`

C Syntax

```
#include "matrix.h"
void mxSetImagData(mxArray *array_ptr, void *pi);
```

Arguments

`array_ptr`
Pointer to an `mxArray`.

`pi`
Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory. If `pi` points to static memory, memory errors will result when the array is destroyed.

Description

`mxSetImagData` is similar to `mxSetPi`, except its `pi` argument is a `void *`. Use this on numeric arrays with contents other than `double`.

Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxSetPi`

Purpose

Set the ir array of a sparse mxArray

C Syntax

```
#include "matrix.h"
void mxSetIr(mxArray *array_ptr, int *ir);
```

Arguments

array_ptr
Pointer to a sparse mxArray.

ir
Pointer to the ir array. The ir array must be sorted in column-major order.

Description

Use mxSetIr to specify the ir array of a sparse mxArray. The ir array is an array of integers; the length of the ir array should equal the value of nzmax.

Each element in the ir array indicates a row (offset by 1) at which a nonzero element can be found. (The jc array is an index that indirectly specifies a column where nonzero elements can be found. See mxSetJc for more details on jc.)

For example, suppose you create a 7-by-3 sparse mxArray named Sparrow containing six nonzero elements by typing

```
Sparrow=zeros(7,3);
Sparrow(2,1)=1;
Sparrow(5,1)=1;
Sparrow(3,2)=1;
Sparrow(2,3)=2;
Sparrow(5,3)=1;
Sparrow(6,3)=1;
Sparrow=sparse(Sparrow);
```

The pr array holds the real data for the sparse matrix, which in Sparrow is the five 1s and the one 2. If there is any nonzero imaginary data, then it is in a pi array.

Subscript	ir	pr	jc	Comments
(2,1)	1	1	0	Column 1; ir is 1 because row is 2.
(5,1)	4	1	2	Column 1; ir is 4 because row is 5.

Subscript	ir	pr	jc	Comments
(3,2)	2	1	3	Column 2; `ir` is 2 because row is 3.
(2,3)	1	2	6	Column 3; `ir` is 1 because row is 2.
(5,3)	4	1		Column 3; `ir` is 4 because row is 5.
(6,3)	5	1		Column 3; `ir` is 5 because row is 6.

Notice how each element of the `ir` array is always 1 less than the row of the corresponding nonzero element. For instance, the first nonzero element is in row 2; therefore, the first element in `ir` is 1 (that is, 2-1). The second nonzero element is in row 5; therefore, the second element in `ir` is 4 (5-1).

The `ir` array must be in column-major order. That means that the `ir` array must define the row positions in column 1 (if any) first, then the row positions in column 2 (if any) second, and so on through column N. Within each column, row position 1 must appear prior to row position 2, and so on.

`mxSetIr` does not sort the `ir` array for you; you must specify an `ir` array that is already sorted.

Examples

See `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `explore.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetJc`

Purpose

Set the `jc` array of a sparse `mxArray`

C Syntax

```
#include "matrix.h"
void mxSetJc(mxArray *array_ptr, int *jc);
```

Arguments

`array_ptr`
Pointer to a sparse `mxArray`.

`jc`
Pointer to the `jc` array.

Description

Use `mxSetJc` to specify a new `jc` array for a sparse `mxArray`. The `jc` array is an integer array having `n+1` elements where `n` is the number of columns in the sparse `mxArray`. The values in the `jc` array have the meanings:

- `jc[j]` is the index in `ir`, `pr` (and `pi` if it exists) of the first nonzero entry in the jth column.
- `jc[j+1]-1` is the index of the last nonzero entry in the jth column.
- `jc`[number of columns `+ 1`] is equal to `nnz`, which is the number of nonzero entries in the entire spare `mxArray`.

The number of nonzero elements in any column (denoted as column C) is

```
jc[C] - jc[C-1];
```

For example, consider a 7-by-3 sparse `mxArray` named `Sparrow` containing six nonzero elements, created by typing

```
Sparrow=zeros(7,3);
Sparrow(2,1)=1;
Sparrow(5,1)=1;
Sparrow(3,2)=1;
Sparrow(2,3)=2;
Sparrow(5,3)=1;
Sparrow(6,3)=1;
Sparrow=sparse(Sparrow);
```

The contents of the `ir`, `jc`, and `pr` arrays are:

Subscript	ir	pr	jc	Comment
(2,1)	1	1	0	Column 1 contains two entries, at `ir[0]`,`ir[1]`
(5,1)	4	1	2	Column 2 contains one entry, at `ir[2]`
(3,2)	2	1	3	Column 3 contains three entries, at `ir[3]`,`ir[4]`, `ir[5]`
(2,3)	1	2	6	There are six nonzero elements.
(5,3)	4	1		
(6,3)	5	1		

As an example of a much sparser `mxArray`, consider an 8,000 element sparse `mxArray` named `Spacious` containing only three nonzero elements. The `ir`, `pr`, and `jc` arrays contain:

Subscript	ir	pr	jc	Comment
(73,2)	72	1	0	Column 1 contains zero entries
(50,3)	49	1	0	Column 2 contains one entry, at `ir[0]`
(64,5)	63	1	1	Column 3 contains one entry, at `ir[1]`
			2	Column 4 contains zero entries.
			2	Column 5 contains one entry, at `ir[3]`
			3	Column 6 contains zero entries.
			3	Column 7 contains zero entries.
			3	Column 8 contains zero entries.
			3	There are three nonzero elements.

Examples See `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `explore.c` in the `mex` subdirectory of the `examples` directory.

See Also `mxGetIr`, `mxGetJc`, `mxSetIr`

mxSetLogical (Obsolete)

Purpose

Convert an `mxArray` to logical type

Note As of MATLAB version 6.5, `mxSetLogical` is obsolete. Support for `mxSetLogical` may be removed in a future version.

C Syntax

```
#include "matrix.h"
void mxSetLogical(mxArray *array_ptr);
```

Arguments

`array_ptr`
Pointer to an `mxArray` having a numeric class.

Description

Use `mxSetLogical` to turn on an `mxArray s` logical flag. This flag tells MATLAB that the array's data is to be treated as Boolean. If the logical flag is on, then MATLAB treats a 0 value as meaning `false` and a nonzero value as meaning `true`. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

Examples

See `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateLogicalScalar`, `mxCreateLogicalMatrix`, `mxCreateLogicalArray`, `mxCreateSparseLogicalMatrix`

Purpose

Set the number of rows

C Syntax

```
#include "matrix.h"
void mxSetM(mxArray *array_ptr, int m);
```

Arguments

`m`
The desired number of rows.

`array_ptr`
Pointer to an `mxArray`.

Description

Call `mxSetM` to set the number of rows in the specified `mxArray`. The term “rows” means the first dimension of an `mxArray`, regardless of the number of dimensions. Call `mxSetN` to set the number of columns.

You typically use `mxSetM` to change the shape of an existing `mxArray`. Note that `mxSetM` does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. Consequently, if your calls to `mxSetM` and `mxSetN` increase the number of elements in the `mxArray`, then you must enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays. Call `mxRealloc` to enlarge them.

If your calls to `mxSetM` and `mxSetN` end up reducing the number of elements in the `mxArray`, then you may want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

Examples

See `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `sincall.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxGetM`, `mxGetN`, `mxSetN`

mxSetN

Purpose

Set the number of columns

C Syntax

```
#include "matrix.h"
void mxSetN(mxArray *array_ptr, int n);
```

Arguments

`array_ptr`
Pointer to an `mxArray`.

`n`
The desired number of columns.

Description

Call `mxSetN` to set the number of columns in the specified `mxArray`. The term "columns" always means the second dimension of a matrix. Calling `mxSetN` forces an `mxArray` to have two dimensions. For example, if `array_ptr` points to an `mxArray` having three dimensions, calling `mxSetN` reduces the `mxArray` to two dimensions.

You typically use `mxSetN` to change the shape of an existing `mxArray`. Note that `mxSetN` does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. Consequently, if your calls to `mxSetN` and `mxSetM` increase the number of elements in the `mxArray`, then you must enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays.

If your calls to `mxSetM` and `mxSetN` end up reducing the number of elements in the `mxArray`, then you may want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

Examples

See `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `sincall.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxGetM`, `mxGetN`, `mxSetM`

V5 Compatible

This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB. If you need to use this function in existing code, use the `-V5` option of the `mex` script.

mxSetNzmax

Purpose Set the storage space for nonzero elements

C Syntax

```
#include "matrix.h"
void mxSetNzmax(mxArray *array_ptr, int nzmax);
```

Arguments

`array_ptr`
Pointer to a sparse `mxArray`.

`nzmax`
The number of elements that `mxCreateSparse` should allocate to hold the arrays pointed to by `ir`, `pr`, and `pi` (if it exists). Set `nzmax` greater than or equal to the number of nonzero elements in the `mxArray`, but set it to be less than or equal to the number of rows times the number of columns. If you specify an `nzmax` value of 0, `mxSetNzmax` sets the value of `nzmax` to 1.

Description

Use `mxSetNzmax` to assign a new value to the `nzmax` field of the specified sparse `mxArray`. The `nzmax` field holds the maximum possible number of nonzero elements in the sparse `mxArray`.

The number of elements in the `ir`, `pr`, and `pi` (if it exists) arrays must be equal to `nzmax`. Therefore, after calling `mxSetNzmax`, you must change the size of the `ir`, `pr`, and `pi` arrays. To change the size of one of these arrays:

1. Call `mxCalloc`, setting `n` to the new value of `nzmax`.
2. Call the ANSI C routine `memcpy` to copy the contents of the old array to the new area allocated in Step 1.
3. Call `mxFree` to free the memory occupied by the old array.
4. Call the appropriate `mxSet` routine (`mxSetIr`, `mxSetPr`, or `mxSetPi`) to establish the new memory area as the current one.

Two ways of determining how big you should make `nzmax` are

- Set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space.
- Make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

Examples See `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetNzmax`

mxSetPi

Purpose

Set new imaginary data for an mxArray

C Syntax

```
#include "matrix.h"
void mxSetPi(mxArray *array_ptr, double *pi);
```

Arguments

`array_ptr`
Pointer to a full (nonsparse) mxArray.

`pi`
Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory. If `pi` points to static memory, memory leaks and other memory errors may result.

Description

Use `mxSetPi` to set the imaginary data of the specified `mxArray`.

Most `mxCreate` functions optionally allocate heap space to hold imaginary data. If you tell an `mxCreate` function to allocate heap space (for example, by setting the `ComplexFlag` to `mxComplex` or by setting `pi` to a non-`NULL` value), then you do not ordinarily use `mxSetPi` to initialize the created `mxArray` s imaginary elements. Rather, you call `mxSetPi` to replace the initial imaginary values with new ones.

Examples

See `mxisfinite.c` and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxSetImagData`, `mxGetPi`, `mxGetPr`, `mxSetPr`

Purpose Set new real data for an mxArray

C Syntax

```
#include "matrix.h"
void mxSetPr(mxArray *array_ptr, double *pr);
```

Arguments

array_ptr
Pointer to a full (nonsparse) mxArray.

pr
Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pr points to static memory, then memory leaks and other memory errors may result.

Description Use mxSetPr to set the real data of the specified mxArray.

All mxCreate calls allocate heap space to hold real data. Therefore, you do not ordinarily use mxSetPr to initialize the real elements of a freshly-created mxArray. Rather, you call mxSetPr to replace the initial real values with new ones.

Examples See mxsetnzmax.c in the mx subdirectory of the examples directory.

See Also mxGetPr, mxGetPi, mxSetPi

Fortran Engine Functions

`engClose`	Quit MATLAB engine session
`engEvalString`	Evaluate expression in `character` array
`engGetArray (Obsolete)`	Use `engGetVariable`
`engGetFull (Obsolete)`	Use `engGetVariable` followed by appropriate `mxGet` routines
`engGetMatrix (Obsolete)`	Use `engGetVariable`
`engGetVariable`	Copy variable from engine workspace
`engOpen`	Start MATLAB engine session
`engOutputBuffer`	Specify buffer for MATLAB output
`engPutArray (Obsolete)`	Use `engPutVariable`
`engPutFull (Obsolete)`	Use `mxCreateDoubleMatrix` and `engPutVariable`
`engPutMatrix (Obsolete)`	Use `engPutVariable`
`engPutVariable`	Put variables into engine workspace

engClose

Purpose

Quit a MATLAB engine session

Fortran Syntax

```
integer*4 function engClose(ep)
integer*4 ep
```

Arguments

`ep`
Engine pointer.

Description

This routine allows you to quit a MATLAB engine session.

`engClose` sends a quit command to the MATLAB engine session and closes the connection. It returns 0 on success, and 1 otherwise. Possible failure includes attempting to terminate a MATLAB engine session that was already terminated.

Example

See `fengdemo.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

engEvalString

Purpose Evaluate expression in `character` array

Fortran Syntax

```
integer*4 function engEvalString(ep, command)
integer*4 ep
character*(*) command
```

Arguments

`ep`
Engine pointer.

`command`
`character` array to execute.

Description

`engEvalString` evaluates the expression contained in `command` for the MATLAB engine session, `ep`, previously started by `engOpen`. It returns a nonzero value if the MATLAB session is no longer running, and zero otherwise.

On UNIX systems, `engEvalString` sends commands to MATLAB by writing down a pipe connected to the MATLAB *`stdin`*. Any output resulting from the command that ordinarily appears on the screen is read back from *`stdout`* into the buffer defined by `engOutputBuffer`.

Example

See `fengdemo.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

Purpose

Read `mxArrays` from a MATLAB engine's workspace

Description

This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `engGetVariable` instead.

engGetFull (Obsolete)

Purpose Read full mxArrays from an engine

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
mp = engGetVariable(ep, name)
m  = mxGetM(mp)
n  = mxGetN(mp)
pr = mxGetPr(mp)
pi = mxGetPi(mp)
mxDestroyArray(mp)
```

instead of

```
engGetFull(ep, name, m, n, pr, pi)
```

See Also engGetVariable, mxGetM, mxGetN, mxGetPr, mxGetPi, mxDestroyArray

Purpose Read `mxArrays` from a MATLAB engine's workspace

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `engGetVariable` instead.

engGetVariable

Purpose Copy a variable from a MATLAB engine's workspace

Fortran Syntax

```
integer*4 function engGetVariable(ep, name)
integer*4 ep
character*(*) name
```

Arguments

`ep`
Engine pointer.

`name`
Name of `mxArray` to get from MATLAB.

Description

`engGetVariable` reads the named `mxArray` from the MATLAB engine session associated with `ep` and returns a pointer to a newly allocated `mxArray` structure, or `0` if the attempt fails. `engGetVariable` fails if the named variable does not exist.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

See Also

`engPutVariable`

Purpose

Start a MATLAB engine session

Fortran Syntax

```
integer*4 function engOpen(startcmd)
integer*4 ep
character*(*) startcmd
```

Arguments

`ep`
Engine pointer.

`startcmd`
`Character` array to start MATLAB process.

Description

This routine allows you to start a MATLAB process to use MATLAB as a computational engine.

`engOpen(startcmd)` starts a MATLAB process using the command specified in `startcmd`, establishes a connection, and returns a unique engine identifier, or 0 if the open fails.

On the UNIX system, if `startcmd` is empty, `engOpen` starts MATLAB on the current host using the command `matlab`. If `startcmd` is a hostname, `engOpen` starts MATLAB on the designated host by embedding the specified hostname string into the larger string:

```
"rsh hostname \"/bin/csh -c 'setenv DISPLAY\
   hostname:0; matlab'\""
```

If `startcmd` is anything else (has white space in it, or nonalphanumeric characters), it is executed literally to start MATLAB.

`engOpen` performs the following steps:

1. Creates two pipes.
2. Forks a new process and sets up the pipes to pass *`stdin`* and *`stdout`* from the child to two file descriptors in the parent.
3. Executes a command to run MATLAB (`rsh` for remote execution).

Example

See `fengdemo.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

engOutputBuffer

Purpose Specify buffer for MATLAB output

Fortran Syntax

```
integer*4 function engOutputBuffer(ep, p)
integer*4 ep
character*n p
```

Arguments

`ep`
Engine pointer.

`p`
Character buffer of length `n`, where `n` is the length of buffer `p`.

Description

`engOutputBuffer` defines a character buffer for `engEvalString` to return any output that would appear on the screen. It returns `1` if you pass it a NULL engine pointer. Otherwise, it returns `0`.

The default behavior of `engEvalString` is to discard any standard output caused by the command it is executing. `engOutputBuffer(ep, p)` tells any subsequent calls to `engEvalString` to save the first `n` characters of output in the character buffer `p`.

Purpose Read `mxArrays` from a MATLAB engine's workspace

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `engPutVariable` instead.

engPutFull (Obsolete)

Purpose

Write full mxArrays into the workspace of an engine

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
mp = mxCreateDoubleMatrix(m, n, 1)
mxSetPr(mp, pr)
mxSetPi(mp, pi)
engPutVariable(ep, name, mp)

mxDestroyArray(mp)
```

instead of

```
engPutFull(ep, name, m, n, pr, pi)
```

See Also

engPutVariable, mxCreateDoubleMatrix, mxSetPr, mxSetPi, mxDestroyArray

Purpose

Write `mxArrays` into a MATLAB engine's workspace

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `engPutVariable` instead.

engPutVariable

Purpose

Put variables into a MATLAB engine's workspace

Fortran Syntax

```
integer*4 function engPutVariable(ep, mp)
integer*4 ep, mp
```

Arguments

`ep`
Engine pointer.

`mp`
`mxArray` pointer.

Description

`engPutVariable` writes `mxArray` `mp` to the engine `ep`. If the `mxArray` does not exist in the workspace, it is created. If an `mxArray` with the same name already exists in the workspace, the existing `mxArray` is replaced with the new `mxArray`.

`engPutVariable` returns `0` if successful and `1` if an error occurs.

See Also

`engGetVariable`

Fortran MAT-File Functions

`matClose`	Close MAT-file
`matDeleteArray (Obsolete)`	Use `matDeleteVariable`
`matDeleteMatrix (Obsolete)`	Use `matDeleteVariable`
`matDeleteVariable`	Delete named `mxArray` from MAT-file
`matGetArray (Obsolete)`	Use `matGetVariable`
`matGetArrayHeader (Obsolete)`	Use `matGetVariableInfo`
`matGetDir`	Get directory of `mxArrays` in MAT-file
`matGetFull (Obsolete)`	Use `matGetVariable` followed by the appropriate `mxGet` routines
`matGetMatrix (Obsolete)`	Use `matGetVariable`
`matGetNextArray (Obsolete)`	Use `matGetNextVariable`
`matGetNextArrayHeader (Obsolete)`	Use `matGetNextVariableInfo`
`matGetNextMatrix (Obsolete)`	Use `matGetNextVariable`
`matGetNextVariable`	Read next `mxArray` from MAT-file
`matGetNextVariableInfo`	Load array header information only
`matGetString (Obsolete)`	Use `matGetVariable` and `mxGetString`
`matGetVariable`	Read `mxArray` from MAT-file
`matGetVariableInfo`	Load array header information only
`matOpen`	Open MAT-file
`matPutArray (Obsolete)`	Use `matPutVariable`

`matPutArrayAsGlobal (Obsolete)`	Use `matPutVariableAsGlobal`
`matPutFull (Obsolete)`	Use `mxCreateDoubleMatrix` and `matPutVariable`
`matPutMatrix (Obsolete)`	Use `matPutVariable`
`matPutString (Obsolete)`	Use `mxCreateString` and `matPutArray`
`matPutVariable`	Write `mxArrays` into MAT-files
`matPutVariableAsGlobal`	Put `mxArrays` into MAT-files

Purpose Closes a MAT-file

Fortran Syntax

```
integer*4 function matClose(mfp)
integer*4 mfp
```

Arguments

`mfp`
Pointer to MAT-file information.

Description

`matClose` closes the MAT-file associated with `mfp`. It returns -1 for a write error, and 0 if successful.

Examples

See `matdemo1.f` and `matdemo2.f` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use this MAT-file routine in a Fortran program.

matDeleteArray (Obsolete)

Purpose Reads `mxArrays` from MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matDeleteVariable` instead.

Purpose

Delete named `mxArray` from MAT-file

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `matDeleteVariable` instead.

matDeleteVariable

Purpose Delete named mxArray from MAT-file

Fortran Syntax

```
integer*4 function matDeleteVariable(mfp, name)
integer*4 mfp
character*(*) name
```

Arguments

mfp
Pointer to MAT-file information.

name
Name of mxArray to delete.

Description

matDeleteVariable deletes the named mxArray from the MAT-file pointed to by mfp. The function returns 0 if successful, and nonzero otherwise.

Purpose

Reads `mxArrays` from MAT-files

Description

This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matGetVariable` instead.

matGetArrayHeader (Obsolete)

Purpose Reads `mxArrays` from MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matGetVariableInfo` instead.

Purpose

Get directory of mxArrays in a MAT-file

Fortran Syntax

```
integer*4 function matGetDir(mfp, num)
integer*4 mfp, num
```

Arguments

mfp
Pointer to MAT-file information.

num
Address of the variable to contain the number of mxArrays in the MAT-file.

Description

This routine allows you to get a list of the names of the mxArrays contained within a MAT-file.

matGetDir returns a pointer to an internal array containing pointers to the names of the mxArrays in the MAT-file pointed to by mfp. The length of the internal array (number of mxArrays in the MAT-file) is placed into num. The internal array is allocated using a single mxCalloc. Use mxFree to free the array when you are finished with it.

matGetDir returns 0 and sets num to a negative number if it fails. If num is zero, mfp contains no mxArrays.

MATLAB variable names can be up to length 32.

Example

See matdemo2.f in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this MAT-file routine in a Fortran program.

matGetFull (Obsolete)

Purpose

Reads full mxArrays from MAT-files

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
pm = matGetVariable(mfp, name)
m  = mxGetM(pm)
n  = mxGetN(pm)
pr = mxGetPr(pm)
pi = mxGetPi(pm)

mxDestroyArray(pm)
```

instead of

```
matGetFull(mfp, name, m, n, pr, pi)
```

See Also

matGetVariable, mxGetM, mxGetN, mxGetPr, mxGetPi, mxDestroyArray

Purpose Reads `mxArrays` from MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `matGetVariable` instead.

matGetNextArray (Obsolete)

Purpose Reads `mxArrays` from MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matGetNextVariable` instead.

Purpose Reads `mxArrays` from MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matGetNextVariableInfo` instead.

matGetNextMatrix (Obsolete)

Purpose Get next `mxArray` from MAT-file

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `matGetNextVariable` instead.

Purpose Read next `mxArray` from MAT-file

Fortran Syntax

```
integer*4 function matGetNextVariable(mfp, name)
integer*4 mfp
character*(*) name
```

Arguments

`mfp`
Pointer to MAT-file information.

`name`
Address of the variable to contain the `mxArray` name.

Description

`matGetNextVariable` allows you to step sequentially through a MAT-file and read all the `mxArrays` in a single pass. The function reads the next `mxArray` from the MAT-file pointed to by `mfp` and returns a pointer to a newly allocated `mxArray` structure. MATLAB returns the name of the `mxArray` in `name`.

Use `matGetNextVariable` immediately after opening the MAT-file with `matOpen` and not in conjunction with other MAT-file routines. Otherwise, the concept of the *next* `mxArray` is undefined.

`matGetNextVariable` returns `0` when the end-of-file is reached or if there is an error condition.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

matGetNextVariableInfo

Purpose Load array header information only

Fortran Syntax

```
integer*4 function matGetNextVariableInfo(mfp, name)
integer*4 mfp
character*(*) name
```

Arguments

`mfp`
Pointer to MAT-file information.

`name`
Address of the variable to contain the `mxArray` name.

Description

`matGetNextVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`, from the file's current file offset. MATLAB returns the name of the `mxArray` in `name`.

If `pr`, `pi`, `ir`, and `jc` are set to nonzero values when loaded with `matGetVariable`, matGetNextVariableInfo sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

matGetString (Obsolete)

Purpose

Copy `character` `mxArrays` from MAT-files

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
pm = matGetVariable(mfp, name)
mxGetString(pm, str, strlen)
```

instead of

```
matGetString(mfp, name, str, strlen)
```

matGetVariable

Purpose Read mxArrays from MAT-files

Fortran Syntax

```
integer*4 function matGetVariable(mfp, name)
integer*4 mfp
character*(*) name
```

Arguments

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray` to get from MAT-file.

Description

This routine allows you to copy an `mxArray` out of a MAT-file.

`matGetVariable` reads the named `mxArray` from the MAT-file pointed to by `mfp` and returns a pointer to a newly allocated `mxArray` structure, or `0` if the attempt fails.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

matGetVariableInfo

Purpose

Load array header information only

Fortran Syntax

```
integer*4 function matGetVariableInfo(mfp, name);
integer*4 mfp
character*(*) name
```

Arguments

`mfp`
Pointer to MAT-file information.

`name`
Name of `mxArray`.

Description

`matGetVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`. It recursively creates the cells/structures through their leaf elements, but does not include `pr`, `pi`, `ir`, and `jc`.

If `pr`, `pi`, `ir`, and `jc` are set to nonzero values when loaded with `matGetVariable`, matGetVariableInfo sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

matOpen

Purpose Opens a MAT-file

Fortran Syntax

```
integer*4 function matOpen(filename, mode)
integer*4 mfp
character*(*) filename, mode
```

Arguments

`filename`
Name of file to open.

`mode`
File opening mode. Legal values for `mode` are:

Table 2-1:

`r`	Opens file for reading only. Determines the current version of the MAT-file by inspecting the files and preserves the current version.
`u`	Opens file for update, both reading and writing, but does not create the file if the file does not exist (equivalent to the `r+` mode of `fopen`). Determines the current version of the MAT-file by inspecting the files and preserves the current version.
`w`	Opens file for writing only. Deletes previous contents, if any.
`w4`	Creates a MATLAB 4 MAT-file.

`mfp`
Pointer to MAT-file information.

Description This routine allows you to open MAT-files for reading and writing.

`matOpen` opens the named file and returns a file handle, or 0 if the open fails.

Examples See `matdemo1.f` and `matdemo2.f` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a Fortran program.

Purpose Reads `mxArrays` from MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matPutVariable` instead.

matPutArrayAsGlobal (Obsolete)

Purpose

Reads `mxArrays` from MAT-files

Description

This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use `matPutVariableAsGlobal` instead.

matPutFull (Obsolete)

Purpose

Writes full mxArrays into MAT-files

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
pm = mxCreateDoubleMatrix(m, n, 1)
mxSetPr(pm, pr)
mxSetPi(pm, pi)
matPutVariable(mfp, name, pm)

mxDestroyArray(pm)
```

instead of

```
matPutFull(mfp, name, m, n, pr, pi)
```

See Also

mxCreateDoubleMatrix, mxSetName (Obsolete), mxSetPr, mxSetPi, matPutVariable, mxDestroyArray

matPutMatrix (Obsolete)

Purpose Writes `mxArrays` into MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `matPutVariable` instead.

Purpose Write character mxArrays into MAT-files

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
pm = mxCreateString(str)
matPutVariable(mfp, name, pm)
mxDestroyArray(pm)
```

instead of

```
matPutString(mfp, name, str)
```

matPutVariable

Purpose

Write mxArrays into MAT-files

Fortran Syntax

```
integer*4 function matPutVariable(mfp, name, pm)
integer*4 mfp, pm
character*(*) name
```

Arguments

mfp
Pointer to MAT-file information.

name
Name of mxArray to put into MAT-file.

pm
mxArray pointer.

Description

This routine allows you to put an mxArray into a MAT-file.

matPutVariable writes mxArray pm to the MAT-file mfp. If the mxArray does not exist in the MAT-file, it is appended to the end. If an mxArray with the same name already exists in the file, the existing mxArray is replaced with the new mxArray by rewriting the file. The size of the new mxArray can be different than the existing mxArray.

matPutVariable returns 0 if successful and nonzero if an error occurs.

Purpose Put mxArrays into MAT-files as originating from the global workspace

Fortran Syntax

```
integer*4 function matPutVariableAsGlobal(mfp, name, pm)
integer*4 mfp, pm
character*(*) name
```

Arguments

mfp
Pointer to MAT-file information.

name
Name of mxArray to put into MAT-file.

pm
mxArray pointer.

Description

This routine allows you to put an mxArray into a MAT-file. matPutVariableAsGlobal is similar to matPutVariable, except the array, when loaded by MATLAB, is placed into the global workspace and a reference to it is set in the local workspace. If you write to a MATLAB 4 format file, matPutVariableAsGlobal will not load it as global, and will act the same as matPutVariable.

matPutVariableAsGlobal writes mxArray pm to the MAT-file mfp. If the mxArray does not exist in the MAT-file, it is appended to the end. If an mxArray with the same name already exists in the file, the existing mxArray is replaced with the new mxArray by rewriting the file. The size of the new mxArray can be different than the existing mxArray.

matPutVariableAsGlobal returns 0 if successful and nonzero if an error occurs.

Fortran MEX-Functions

`mexAtExit`	Register function to be called when MATLAB is cleared or terminates
`mexCallMATLAB`	Call MATLAB function or user-defined M-file or MEX-file
`mexErrMsgIdAndTxt`	Issue error message with identifier and return to MATLAB
`mexErrMsgTxt`	Issue error message and return to MATLAB
`mexEvalString`	Execute MATLAB command in caller's workspace
`mexFunction`	Entry point to Fortran MEX-file
`mexFunctionName`	Name of current MEX-function
`mexGetArray (Obsolete)`	Use `mexGetVariable`
`mexGetArrayPtr (Obsolete)`	Use `mexGetVariablePtr`
`mexGetEps (Obsolete)`	Use `mxGetEps`
`mexGetFull (Obsolete)`	Use `mexGetVariable`, `mxGetM`, `mxGetN`, `mxGetPr`, `mxGetPi`
`mexGetGlobal (Obsolete)`	Use `mexGetVariablePtr`
`mexGetInf (Obsolete)`	Use `mxGetInf`
`mexGetMatrix (Obsolete)`	Use `mexGetVariable`
`mexGetMatrixPtr (Obsolete)`	Use `mexGetVariablePtr`
`mexGetNaN (Obsolete)`	Use `mxGetNaN`

`mexGetVariable`	Get copy of variable from another workspace
`mexGetVariablePtr`	Get read-only pointer to variable from another workspace
`mexIsFinite (Obsolete)`	Use `mxIsFinite`
`mexIsGlobal`	True if mxArray has global scope
`mexIsInf (Obsolete)`	Use `mxIsInf`
`mexIsLocked`	True if MEX-file is locked
`mexIsNaN (Obsolete)`	Use `mxIsNaN`
`mexLock`	Lock MEX-file so it cannot be cleared from memory
`mexMakeArrayPersistent`	Make `mxArray` persist after MEX-file completes
`mexMakeMemoryPersistent`	Make memory allocated by MATLAB memory allocation routines persist after MEX-file completes
`mexPrintf`	ANSI C printf-style output routine
`mexPutArray (Obsolete)`	Use `mexPutVariable`
`mexPutFull (Obsolete)`	Use `mxCreateDoubleMatrix`, `mxSetPr`, `mxSetPi`, `mexPutVariable`
`mexPutMatrix (Obsolete)`	Use `mexPutVariable`
`mexPutVariable`	Copy mxArray from your MEX-file into another workspace
`mexSetTrapFlag`	Control response of `mexCallMATLAB` to errors
`mexUnlock`	Unlock MEX-file so it can be cleared from memory
`mexWarnMsgIdAndTxt`	Issue warning message with identifier
`mexWarnMsgTxt`	Issue warning message

Purpose Register a subroutine to be called when the MEX-file is cleared or when MATLAB terminates

Fortran Syntax

```
integer*4 function mexAtExit(ExitFcn)
subroutine ExitFcn()
```

Arguments

`ExitFcn`
The exit function. This function must be declared as `external`.

Returns Always returns 0.

Description Use `mexAtExit` to register a subroutine to be called just before the MEX-file is cleared or MATLAB is terminated. `mexAtExit` gives your MEX-file a chance to perform an orderly shutdown of anything under its control.

Each MEX-file can register only one active exit subroutine at a time. If you call `mexAtExit` more than once, MATLAB uses the `ExitFcn` from the more recent `mexAtExit` call as the exit function.

If a MEX-file is locked, all attempts to clear the MEX-file will fail. Consequently, if a user attempts to clear a locked MEX-file, MATLAB does not call the `ExitFcn`.

You must declare the `ExitFcn` as `external` in the Fortran routine that calls `mexAtExit` if it is not within the scope of the file.

See Also `mexSetTrapFlag`

mexCallMATLAB

Purpose Call a MATLAB function or operator, a user-defined M-file, or other MEX-file

Fortran Syntax

```
integer*4 function mexCallMATLAB(nlhs, plhs, nrhs, prhs, name)
integer*4 nlhs, nrhs, plhs(*), prhs(*)
character*(*) name
```

On the Alpha platform, use:

```
integer*8 function mexCallMATLAB(nlhs, plhs, nrhs, prhs, name)
integer*4 nlhs, nrhs
integer*8 plhs(*), prhs(*)
character*(*) name
```

Arguments

`nlhs`
Number of desired output arguments. This value must be less than or equal to 50.

`plhs`
Array of `mxArray` pointers that can be used to access the returned data from the function call. Once the data is accessed, you can then call `mxFree` to free the `mxArray` pointer. By default, MATLAB frees the pointer and any associated dynamic memory it allocates when you return from the `mexFunction` call.

`nrhs`
Number of input arguments. This value must be less than or equal to 50.

`prhs`
Array of pointers to input data.

`name`
`Character` array containing the name of the MATLAB function, operator, M-file, or MEX-file that you are calling. If `name` is an operator, place the operator inside a pair of single quotes; for example, `'+'`.

Returns 0 if successful, and a nonzero value if unsuccessful and `mexSetTrapFlag` was previously called.

Description Call `mexCallMATLAB` to invoke internal MATLAB functions, MATLAB operators, M-files, or other MEX-files.

By default, if `name` detects an error, MATLAB terminates the MEX-file and returns control to the MATLAB prompt. If you want a different error behavior, turn on the trap flag by calling `mexSetTrapFlag`.

See Also `mexFunction`, `mexSetTrapFlag`

mexErrMsgIdAndTxt

Purpose Issue error message with identifier and return to the MATLAB prompt

Fortran Syntax

```
subroutine mexErrMsgIdAndTxt(errorid, errormsg)
character*(*) errorid, errormsg
```

Arguments

`errorid`
Character array containing a MATLAB message identifier. See "Message Identifiers" in the MATLAB documentation for information on this topic.

`errormsg`
`Character` array containing the error message to be displayed.

Description

Call `mexErrMsgIdAndTxt` to write an error message and its corresponding identifier to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgIdAndTxt` does not clear the MEX-file from memory. Consequently, `mexErrMsgIdAndTxt` does not invoke any registered exit routine to allocate memory.

If your application calls `mxCalloc` or one of the `mxCreate` routines to create `mxArray` pointers, `mexErrMsgIdAndTxt` automatically frees any associated memory allocated by these calls.

See Also `mexErrMsgTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

Purpose Issue error message and return to the MATLAB prompt

Fortran Syntax

```
subroutine mexErrMsgTxt(errormsg)
character*(*) errormsg
```

Arguments

`errormsg`
`Character` array containing the error message to be displayed.

Description

Call `mexErrMsgTxt` to write an error message to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgTxt` does not clear the MEX-file from memory. Consequently, `mexErrMsgTxt` does not invoke any registered exit routine to allocate memory.

If your application calls `mxCalloc` or one of the `mxCreate` routines to create `mxArray` pointers, `mexErrMsgTxt` automatically frees any associated memory allocated by these calls.

See Also `mexErrMsgIdAndTxt`, `mexWarnMsgTxt`, `mexWarnMsgIdAndTxt`

mexEvalString

Purpose Execute a MATLAB command in the workspace of the caller

Fortran Syntax

```
integer*4 function mexEvalString(command)
character*(*) command
```

Arguments

`command`
A `character` array containing the MATLAB command to execute.

Returns 0 if successful, and a nonzero value if unsuccessful.

Description Call `mexEvalString` to invoke a MATLAB command in the workspace of the caller.

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. However, `mexCallMATLAB` provides a mechanism for returning results (left-hand side arguments) back to the MEX-file; `mexEvalString` provides no way for return values to be passed back to the MEX-file.

All arguments that appear to the right of an equals sign in the `command` array must already be current variables of the caller's workspace.

See Also `mexCallMATLAB`

Purpose

MATLAB entry point to a Fortran MEX-file

Fortran Syntax

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer*4 nlhs, nrhs, plhs(*), prhs(*)
```

Arguments

`nlhs`
The number of expected outputs.

`plhs`
Array of pointers to expected outputs.

`nrhs`
The number of inputs.

`prhs`
Array of pointers to input data. The input data is read only and should not be altered by your `mexFunction`.

Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of a subroutine you must write in every MEX-file. When you invoke a MEX-file, MATLAB searches for a subroutine named `mexFunction` inside the MEX-file. If it finds one, then the first executable line in `mexFunction` becomes the starting point of the MEX-file. If MATLAB cannot find a subroutine named `mexFunction` inside the MEX-file, MATLAB issues an error message.

When you invoke a MEX-file, MATLAB automatically loads `nlhs`, `plhs`, `nrhs`, and `prhs` with the caller's information. In the syntax of the MATLAB language, functions have the general form

```
[a,b,c, ] = fun(d,e,f, )
```

where the denotes more items of the same format. The `a,b,c` are left-hand side arguments and the `d,e,f` are right-hand side arguments. The arguments `nlhs` and `nrhs` contain the number of left-hand side and right-hand side arguments, respectively, with which the MEX-file is called. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is a pointer to an array whose length is `nlhs`, where your function must set pointers for the returned left-hand side `mxArrays`.

mexFunctionName

Purpose Get the name of the current MEX-function

Fortran Syntax

```
character*(*) function mexFunctionName()
```

Arguments None

Returns The name of the current MEX-function.

Description `mexFunctionName` returns the name of the current MEX-function.

Purpose Get a copy of a variable from the specified workspace

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use

```
mexGetVariable(workspace, name)
```

instead of

```
mexGetArray(rname, workspace)
```

See Also `mexGetVariable`

mexGetArrayPtr (Obsolete)

Purpose Get a read-only pointer to a variable from the specified workspace

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use

```
mexGetVariablePtr(varname, workspace)
```

instead of

```
mexGetArrayPtr(varname, workspace)
```

See Also `mexGetVariable`

Purpose Get the value of `eps`

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxGetEps` instead.

mexGetFull (Obsolete)

Purpose

Routine to get component parts of a double-precision `mxArray` into a Fortran workspace

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
pm = mexGetVariable("caller", name)
m = mxGetM(pm)
n = mxGetN(pm)
pr = mxGetPr(pm)
pi = mxGetPi(pm)
```

instead of

```
mexGetFull(name, m, n, pr, pi)
```

See Also

`mexGetVariable`, `mxGetM`, `mxGetN`, `mxGetPr`, `mxGetPi`

Purpose

Get a pointer to an `mxArray` from the MATLAB global workspace

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
mexGetVariablePtr(name, "global")
```

instead of

```
mexGetGlobal(name)
```

See Also

`mexGetVariablePtr`, `mxGetPr`, `mxGetPi`

mexGetInf (Obsolete)

Purpose Get the value of infinity

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxGetInf` instead.

Purpose Copies an `mxArray` from the caller's workspace

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
mexGetVariable("caller", name)
```

instead of

```
mexGetMatrix(name)
```

See Also `mexGetVariable`

mexGetMatrixPtr (Obsolete)

Purpose

Get the pointer to an `mxArray` in the caller's workspace

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
mexGetVariablePtr(name, "caller")
```

instead of

```
mexGetMatrixPtr(name)
```

See Also

`mexGetVariablePtr`

Purpose Get the value of `NaN` (Not-a-Number)

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxGetNaN` instead.

mexGetVariable

Purpose Get a copy of a variable from the specified workspace

Fortran Syntax

```
integer*4 function mexGetVariable(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`
Specifies where `mexGetVariable` should search in order to find variable `varname`. The possible values are:

`base`	Search for the variable in the base workspace
`caller`	Search for the variable in the caller's workspace
`global`	Search for the variable in the global workspace

`varname`
Name of the variable to copy.

Returns A copy of the variable on success. Returns `0` on failure. A common cause of failure is specifying a variable that is not currently in the workspace.

Description Call `mexGetVariable` to get a copy of the specified variable. The returned `mxArray` contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned `mxArray` do not affect the variable in the workspace unless you write the copy back to the workspace with `mexPutVariable`.

See Also `mexGetVariablePtr`, `mexPutVariable`

Purpose

Get a read-only pointer to a variable from the specified workspace

Fortran Syntax

```
integer*4 function mexGetVariablePtr(varname, workspace)
character*(*) varname, workspace
```

Arguments

`varname`
Name of the variable to copy. (Note that this is a variable name, not an `mxArray` pointer.)

`workspace`
Specifies which workspace you want `mexGetVariablePtr` to search. The possible values are:

`base`	Search for the variable in the base workspace
`caller`	Search for the variable in the caller's workspace
`global`	Search for the variable in the global workspace

Returns

A read-only pointer to the `mxArray` on success. Returns `0` on failure.

Description

Call `mexGetVariablePtr` to get a read-only pointer to the specified variable `varname` from the specified workspace. This command is useful for examining an `mxArray`'s data and characteristics. If you need to change data or characteristics, use `mexGetVariable` (along with `mexPutVariable`) instead of `mexGetVariablePtr`.

See Also

`mexGetVariable`

mexIsFinite (Obsolete)

Purpose

Determine whether or not a value is finite

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxIsFinite` instead.

Purpose

True if mxArray has global scope

Fortran Syntax

```
integer*4 function mexIsGlobal(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns

1 if the mxArray has global scope, and 0 otherwise.

Description

Use mexIsGlobal to determine if the specified mxArray has global scope.

See Also

mexGetVariable, mexGetVariablePtr, mexPutVariable, global

mexIsInf (Obsolete)

Purpose Determine whether or not a value is infinite

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxIsInf` instead.

Purpose Determine if this MEX-file is locked

Fortran Syntax

```
integer*4 function mexIsLocked()
```

Arguments `none`

Returns `1` if the MEX-file is locked; `0` if the file is unlocked.

Description Call `mexIsLocked` to determine if the MEX-file is locked. By default, MEX-files are unlocked, meaning that users can clear the MEX-file at any time.

To unlock a MEX-file, call `mexUnlock`.

See Also `mexLock`, `mexUnlock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

mexIsNaN (Obsolete)

Purpose Determine whether or not a value is `NaN` (Not-a-Number)

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxIsNaN` instead.

Purpose Lock a MEX-file so that it cannot be cleared from memory

Fortran Syntax

```
subroutine mexLock()
```

Arguments `none`

Description By default, MEX-files are unlocked, meaning that a user can clear them at any time. Call `mexLock` to prohibit a MEX-file from being cleared.

To unlock a MEX-file, call `mexUnlock`.

`mexLock` increments a lock count. If you call `mexLock` `n` times, you must call `mexUnlock` `n` times to unlock your MEX-file.

See Also `mexIsLocked`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

mexMakeArrayPersistent

Purpose

Make an mxArray persist after the MEX-file completes

Fortran Syntax

```
subroutine mexMakeArrayPersistent(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray created by an mxCreate* routine.

Description

By default, mxArrays allocated by mxCreate* routines are not persistent. The MATLAB memory management facility automatically frees nonpersistent mxArrays when the MEX-file finishes. If you want the mxArray to persist through multiple invocations of the MEX-file, you must call mexMakeArrayPersistent.

Note If you create a persistent mxArray, you are responsible for destroying it when the MEX-file is cleared. If you do not destroy a persistent mxArray, MATLAB will leak memory. See mexAtExit on how to register a function that gets called when the MEX-file is cleared. See mexLock on how to lock your MEX-file so that it is never cleared.

See Also

mexAtExit, mexLock, mexMakeMemoryPersistent, and the mxCreate functions.

Purpose

Make memory allocated by MATLAB memory allocation routines (`mxCalloc`, `mxMalloc`, `mxRealloc`) persist after the MEX-file completes

Fortran Syntax

```
subroutine mexMakeMemoryPersistent(ptr)
integer*4 ptr
```

Arguments

`ptr`
Pointer to the beginning of memory allocated by one of the MATLAB memory allocation routines.

Description

By default, memory allocated by MATLAB is nonpersistent, so it is freed automatically when the MEX-file finishes. If you want the memory to persist, you must call `mexMakeMemoryPersistent`.

Note If you create persistent memory, you are responsible for freeing it when the MEX-file is cleared. If you do not free the memory, MATLAB will leak memory. To free memory, use `mxFree`. See `mexAtExit` on how to register a function that gets called when the MEX-file is cleared. See `mexLock` on how to lock your MEX-file so that it is never cleared.

See Also

`mexAtExit`, `mexLock`, `mexMakeArrayPersistent`, `mxCalloc`, `mxFree`, `mxMalloc`, `mxRealloc`

mexPrintf

Purpose Print a `character` array

Fortran Syntax

```
integer*4 function mexPrintf(message)
character*(*) message
```

Arguments

`message`
Character array containing message to be displayed.

Note Optional arguments to `mexPrintf`, such as format strings, are not supported in Fortran.

Note If you want the literal `%` in your message, you must use `%%` in your message string since `%` has special meaning to `mexPrintf`. Failing to do so causes unpredictable results.

Returns The number of characters printed. This includes characters specified with backslash codes, such as `\n` and `\b`.

Description `mexPrintf` prints a `character` array on the screen and in the diary (if the diary is in use). It provides a callback to the standard C `printf` routine already linked inside MATLAB.

See Also `mexErrMsgTxt`

Purpose Copy an `mxArray` into the specified workspace

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Use

```
mexPutVariable(workspace, name, pm)
```

instead of

```
mexPutArray(pm, workspace)
```

See Also `mexPutVariable`

mexPutFull (Obsolete)

Purpose

Routine to create an `mxArray` from its component parts into a Fortran workspace

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
pm = mxCreateDoubleMatrix(m, n, 1)
mxSetPr(pm, pr)
mxSetPi(pm, pi)
mexPutVariable("caller", name, pm)
```

instead of

```
mexPutFull(name, m, n, pr, pi)
```

See Also

`mxCreateDoubleMatrix`, `mxSetName (Obsolete)`, `mxSetPr`, `mxSetPi`, `mexPutVariable`

Purpose Writes an `mxArray` to the caller's workspace

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
mexPutVariable("caller", name, pm)
```

instead of

```
mexPutMatrix(pm)
```

mexPutVariable

Purpose

Copy an mxArray into the specified workspace

Fortran Syntax

```
integer*4 function mexPutVariable(workspace, varname, pm)
character*(*) workspace, varname
integer*4 pm
```

Arguments

workspace
Specifies the scope of the array that you are copying. The possible values are:

base	Copy the mxArray to the base workspace
caller	Copy the mxArray to the caller's workspace
global	Copy the mxArray to the list of global variables

varname
Name given to the mxArray in the workspace.

pm
Pointer to an mxArray.

Returns

0 on success; 1 on failure. A possible cause of failure is that the pm argument is zero.

Description

Call mexPutVariable to copy the mxArray, at pointer pm, from your MEX-file into the specified workspace. MATLAB gives the name, varname, to the copied mxArray in the receiving workspace.

mexPutVariable makes the array accessible to other entities, such as MATLAB, M-files or other MEX-files.

If a variable of the same name already exists in the specified workspace, mexPutVariable overwrites the previous contents of the variable with the contents of the new mxArray. For example, suppose the MATLAB workspace defines variable Peaches as

```
Peaches
1      2      3      4
```

and you call mexPutVariable to copy Peaches into the MATLAB workspace.

```
mexPutVariable("base", "Peaches", pm)
```

Then the old value of `Peaches` disappears and is replaced by the value passed in by `mexPutVariable`.

See Also

`mexGetVariable`

mexSetTrapFlag

Purpose Control response of `mexCallMATLAB` to errors

Fortran Syntax

```
subroutine mexSetTrapFlag(trapflag)
integer*4 trapflag
```

Arguments

`trapflag`
Control flag. Currently, the only legal values are:

0	On error, control returns to the MATLAB prompt.
1	On error, control returns to your MEX-file.

Description Call `mexSetTrapFlag` to control the MATLAB response to errors in `mexCallMATLAB`.

If you do not call `mexSetTrapFlag`, then whenever MATLAB detects an error in a call to `mexCallMATLAB`, MATLAB automatically terminates the MEX-file and returns control to the MATLAB prompt. Calling `mexSetTrapFlag` with `trapflag` set to `0` is equivalent to not calling `mexSetTrapFlag` at all.

If you call `mexSetTrapFlag` and set the `trapflag` to `1`, then whenever MATLAB detects an error in a call to `mexCallMATLAB`, MATLAB does not automatically terminate the MEX-file. Rather, MATLAB returns control to the line in the MEX-file immediately following the call to `mexCallMATLAB`. The MEX-file is then responsible for taking an appropriate response to the error.

See Also `mexAtExit`, `mexErrMsgTxt`

Purpose

Unlock this MEX-file so that it can be cleared from memory

Fortran Syntax

```
subroutine mexUnlock()
```

Arguments

`none`

Description

By default, MEX-files are unlocked, meaning that a user can clear them at any time. Calling `mexLock` locks a MEX-file so that it cannot be cleared. Calling `mexUnlock` removes the lock so that the MEX-file can be cleared.

`mexLock` increments a lock count. If you called `mexLock` `n` times, you must call `mexUnlock` `n` times to unlock your MEX-file.

See Also

`mexIsLocked`, `mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

mexWarnMsgIdAndTxt

Purpose Issue warning message with identifier

Fortran Syntax

```
subroutine mexWarnMsgIdAndTxt(warningid, warningmsg)
character*(*) warningid, warningmsg
```

Arguments

`errorid`
Character array containing a MATLAB message identifier. See "Message Identifiers" in the MATLAB documentation for information on this topic.

`warningmsg`
String containing the warning message to be displayed.

Description

`mexWarnMsgIdAndTxt` causes MATLAB to display the contents of `warningmsg`.

Unlike `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt` does not cause the MEX-file to terminate.

See Also

`mexWarnMsgTxt`, `mexErrMsgIdAndTxt`, `mexErrMsgTxt`

Purpose Issue warning message

Fortran Syntax

```
subroutine mexWarnMsgTxt(warningmsg)
character*(*) warningmsg
```

Arguments

`warningmsg`
String containing the warning message to be displayed.

Description

`mexWarnMsgTxt` causes MATLAB to display the contents of `warningmsg`.

Unlike `mexErrMsgTxt`, `mexWarnMsgTxt` does not cause the MEX-file to terminate.

See Also

`mexWarnMsgIdAndTxt`, `mexErrMsgTxt`, `mexErrMsgIdAndTxt`

8

Fortran MX-Functions

`mxAddField`	Add field to structure array
`mxCalcSingleSubscript`	Return offset from first element to desired element
`mxCalloc`	Allocate dynamic memory using the MATLAB memory manager
`mxClassIDFromClassName`	Get identifier that corresponds to a class
`mxClearLogical (Obsolete)`	Clear logical flag
`mxCopyCharacterToPtr`	Copy `character` values from Fortran array to pointer array
`mxCopyComplex8ToPtr`	Copy `COMPLEX*8` values from Fortran array to pointer array
`mxCopyComplex16ToPtr`	Copy `COMPLEX*16` values from Fortran array to pointer array
`mxCopyInteger1ToPtr`	Copy `INTEGER*1` values from Fortran array to pointer array
`mxCopyInteger2ToPtr`	Copy `INTEGER*2` values from Fortran array to pointer array
`mxCopyInteger4ToPtr`	Copy `INTEGER*4` values from Fortran array to pointer array
`mxCopyPtrToCharacter`	Copy `character` values from pointer array to Fortran array

mxCopyPtrToComplex8	Copy COMPLEX*8 values from pointer array to Fortran array
mxCopyPtrToComplex16	Copy COMPLEX*16 values from pointer array to Fortran array
mxCopyPtrToInteger1	Copy INTEGER*1 values from pointer array to Fortran array
mxCopyPtrToInteger2	Copy INTEGER*2 values from pointer array to Fortran array
mxCopyPtrToInteger4	Copy INTEGER*4 values from pointer array to Fortran array
mxCopyPtrToPtrArray	Copy pointer values from pointer array to Fortran array
mxCopyPtrToReal4	Copy REAL*4 values from pointer array to Fortran array
mxCopyPtrToReal8	Copy REAL*8 values from pointer array to Fortran array
mxCopyReal4ToPtr	Copy REAL*4 values from Fortran array to pointer array
mxCopyReal8ToPtr	Copy REAL*8 values from Fortran array to pointer array
mxCreateCellArray	Create unpopulated N-dimensional cell mxArray
mxCreateCellMatrix	Create unpopulated two-dimensional cell mxArray
mxCreateCharArray	Create unpopulated N-dimensional string mxArray
mxCreateCharMatrixFromStrings	Create populated two-dimensional string mxArray
mxCreateDoubleMatrix	Create unpopulated two-dimensional, double-precision, floating-point mxArray

`mxCreateFull (Obsolete)`	Create unpopulated two-dimensional `mxArray`
`mxCreateNumericArray`	Create unpopulated N-dimensional numeric `mxArray`
`mxCreateNumericMatrix`	Create numeric matrix and initialize data elements to 0
`mxCreateScalarDouble`	Create scalar, double-precision array initialized to specified value
`mxCreateSparse`	Create two-dimensional unpopulated sparse `mxArray`
`mxCreateString`	Create 1-by-n `character` array initialized to specified string
`mxCreateStructArray`	`Create unpopulated N-dimensional structure mxArray`
`mxCreateStructMatrix`	`Create unpopulated two-dimensional structure mxArray`
`mxDestroyArray`	Free dynamic memory allocated by an `mxCreate` routine
`mxDuplicateArray`	Make deep copy of array
`mxFree`	Free dynamic memory allocated by `mxCalloc`
`mxFreeMatrix (Obsolete)`	Free dynamic memory allocated by `mxCreateFull` and `mxCreateSparse`
`mxGetCell`	Get cell's contents
`mxGetClassID`	Get `mxArray`'s class
`mxGetClassName`	Get `mxArray`'s class
`mxGetData`	Get pointer to data
`mxGetDimensions`	Get pointer to dimensions array

`mxGetElementSize`	Get number of bytes required to store each data element
`mxGetEps`	Get value of `eps`
`mxGetField`	Get field value, given field name and index in structure array
`mxGetFieldByNumber`	Get field value, given field number and index in structure array
`mxGetFieldNameByNumber`	Get field name, given field number in structure array
`mxGetFieldNumber`	Get field number, given field name in structure array
`mxGetImagData`	Get pointer to imaginary data of `mxArray`
`mxGetInf`	Get value of infinity
`mxGetIr`	Get `ir` array
`mxGetJc`	Get `jc` array
`mxGetM`	Get number of rows
`mxGetN`	Get total number of columns
`mxGetName (Obsolete)`	Get name of specified `mxArray`
`mxGetNaN`	Get the value of `NaN`
`mxGetNumberOfDimensions`	Get number of dimensions
`mxGetNumberOfElements`	Get number of elements in array
`mxGetNumberOfFields`	Get number of fields in structure `mxArray`
`mxGetNzmax`	Get number of elements in `ir`, `pr`, and `pi` arrays
`mxGetPi`	Get `mxArray`'s imaginary data elements
`mxGetPr`	Get `mxArray`'s real data elements

`mxGetScalar`	Get real component of `mxArray`'s first data element
`mxGetString`	Create `character` array from `mxArray`
`mxIsCell`	True if cell `mxArray`
`mxIsChar`	True if string `mxArray`
`mxIsClass`	True if `mxArray` is member of specified class
`mxIsComplex`	Inquire if `mxArray` is complex
`mxIsDouble`	Inquire if `mxArray` is of type `double`
`mxIsEmpty`	True if `mxArray` is empty
`mxIsFinite`	True if value is finite
`mxIsFromGlobalWS`	True if `mxArray` was copied from the MATLAB global workspace
`mxIsFull (Obsolete)`	Inquire if `mxArray` is full
`mxIsInf`	True if value is infinite
`mxIsInt8`	True if `mxArray` represents its data as signed 8-bit integers
`mxIsInt16`	True if `mxArray` represents its data as signed 16-bit integers
`mxIsInt32`	True if `mxArray` represents its data as signed 32-bit integers
`mxIsLogical`	True if `mxArray` is Boolean
`mxIsNaN`	True if value is `NaN`
`mxIsNumeric`	Inquire if `mxArray` contains numeric data
`mxIsSingle`	True if `mxArray` represents its data as single-precision, floating-point numbers
`mxIsSparse`	Inquire if `mxArray` is sparse

`mxIsString (Obsolete)`	Inquire if `mxArray` contains `character` array
`mxIsStruct`	True if structure `mxArray`
`mxIsUint8`	True if `mxArray` represents its data as unsigned 8-bit integers
`mxIsUint16`	True if `mxArray` represents its data as unsigned 16-bit integers
`mxIsUint32`	True if `mxArray` represents its data as unsigned 32-bit integers
`mxMalloc`	Allocate dynamic memory using the MATLAB memory manager
`mxRealloc`	Reallocate memory
`mxRemoveField`	Remove field from structure array
`mxSetCell`	Set value of one cell
`mxSetData`	Set pointer to data
`mxSetDimensions`	Modify number/size of dimensions
`mxSetField`	Set field value of structure array, given field name/index
`mxSetFieldByNumber`	Set field value in structure array, given field number/index
`mxSetImagData`	Set imaginary data pointer for `mxArray`
`mxSetIr`	Set `ir` array of sparse `mxArray`
`mxSetJc`	Set `jc` array of sparse `mxArray`
`mxSetLogical (Obsolete)`	Set logical flag
`mxSetM`	Set number of rows
`mxSetN`	Set number of columns
`mxSetName (Obsolete)`	Set name of `mxArray`

`mxSetNzmax`	Set storage space for nonzero elements
`mxSetPi`	Set new imaginary data for an mxArray
`mxSetPr`	Set new real data for an mxArray

Purpose Add a field to a structure array

Fortran Syntax

```
integer*4 function mxAddField(pm, fieldname)
integer*4 pm
character*(*) fieldname
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`fieldname`
The name of the field you want to add.

Returns Field number on success, or `0` if inputs are invalid or an out-of-memory condition occurs.

Description Call `mxAddField` to add a field to a structure array. You must then create the values with the `mxCreate*` functions and use `mxSetFieldByNumber` to set the individual values for the field.

See Also `mxRemoveField`, `mxSetFieldByNumber`

mxCalcSingleSubscript

Purpose

Return the offset (index) from the first element to the desired element

Fortran Syntax

```
integer*4 function mxCalcSingleSubscript(pm, nsubs, subs)
integer*4 pm, nsubs, subs
```

Arguments

`pm`
Pointer to an `mxArray`.

`nsubs`
The number of elements in the `subs` array. Typically, you set `nsubs` equal to the number of dimensions in the `mxArray` that `pm` points to.

`subs`
An array of integers. Each value in the array should specify that dimension's subscript. The value in `subs(1)` specifies the row subscript, and the value in `subs(2)` specifies the column subscript. Use 1-based indexing to specify the desired array element. For example, to express the starting element of a two-dimensional `mxArray` in `subs`, set `subs(1)` to 1 and `subs(2)` to `1`.

Returns

The number of elements between the start of the `mxArray` and the specified subscript. This returned number is called an "index"; many `mx` routines (for example, `mxGetField`) require an index as an argument.

If `subs` describes the starting element of an `mxArray`, `mxCalcSingleSubscript` returns 0. If `subs` describes the final element of an `mxArray`, then `mxCalcSingleSubscript` returns `N-1` (where `N` is the total number of elements).

Description

Call `mxCalcSingleSubscript` to determine how many elements there are between the beginning of the `mxArray` and a given element of that `mxArray`. For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the `(1,1)` element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have.

Use `mxCalcSingleSubscript` with functions that interact with multidimensional cells and structures. `mxGetCell` and `mxSetCell` are two such functions.

See Also

`mxGetCell`, `mxSetCell`

Purpose

Allocate dynamic memory using the MATLAB memory manager

Fortran Syntax

```
integer*4 function mxCalloc(n, size)
integer*4 n, size
```

Arguments

`n`
Number of elements to allocate. This must be a nonnegative number.

`size`
Number of bytes per element.

Returns

A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCalloc` returns 0. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.

`mxCalloc` is unsuccessful when there is insufficient free heap space.

Description

The MATLAB memory management facility maintains a list of all memory allocated by `mxCalloc` (and by the `mxCreate` calls). The MATLAB memory management facility automatically frees (deallocates) all of a MEX-file's parcels when control returns to the MATLAB prompt.

By default, in a MEX-file, `mxCalloc` generates nonpersistent `mxCalloc` data. In other words, the memory management facility automatically deallocates the memory as soon as the MEX-file ends. When you finish using the memory allocated by `mxCalloc`, call `mxFree`. `mxFree` deallocates the memory.

`mxCalloc` works differently in MEX-files than in stand-alone MATLAB applications. In MEX-files, `mxCalloc` automatically

- Allocates enough contiguous heap space to hold `n` elements.
- Initializes all `n` elements to 0.
- Registers the returned heap space with the MATLAB memory management facility.

In stand-alone MATLAB applications, the MATLAB memory manager is not used.

See Also

`mxFree`

mxClassIDFromClassName

Purpose

Get identifier that corresponds to a class

Fortran Syntax

```
integer*4 function mxClassIDFromClassName(classname)
character*(*) classname
```

Arguments

classname
A character array specifying a MATLAB class name. Use one of the strings from the table below.

Returns

A numeric identifier used internally by MATLAB to represent the MATLAB class, *classname*. Returns 0 if *classname* is not a recognized MATLAB class.

Description

Use mxClassIDFromClassName to obtain an identifier for any class that is recognized by MATLAB. This function is most commonly used to provide a classid argument to mxCreateNumericArray and mxCreateNumericMatrix.

Valid choices for *classname* are shown below. MATLAB returns 0 if *classname* is unrecognized.

cell	char	double	function_handle
int8	int16	int32	int64
object	single	sparse	struct
uint8	uint16	uint32	uint64

See Also

mxGetClassName, mxCreateNumericArray, mxCreateNumericMatrix

Purpose

Clear the logical flag

Note As of MATLAB version 6.5, `mxClearLogical` is obsolete. Support for `mxClearLogical` may be removed in a future version.

Fortran Syntax

```
subroutine mxClearLogical(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray` having a numeric class.

Description

Use `mxClearLogical` to turn off the `mxArray`'s logical flag. This flag, when cleared, tells MATLAB that the `mxArray`'s data is to be treated as numeric data rather than as Boolean data. If the logical flag is on, then MATLAB treats a 0 value as meaning `false` and a nonzero value as meaning `true`.

Call `mxSetLogical` to turn on the `mxArray`'s logical flag. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

See Also

`mxIsLogical`, `mxSetLogical (Obsolete)`, `logical`

mxCopyCharacterToPtr

Purpose Copy `character` values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyCharacterToPtr(y, px, n)
character*(*) y
integer*4 px, n
```

Arguments

`y`
`character` Fortran array.

`px`
Pointer to `character` or `name` array.

`n`
Number of elements to copy.

Description `mxCopyCharacterToPtr` copies `n` `character` values from the Fortran `character` array `y` into the MATLAB string array pointed to by `px`. This subroutine is essential for copying character data between MATLAB pointer arrays and ordinary Fortran `character` arrays.

See Also `mxCopyPtrToCharacter`, `mxCreateCharArray`, `mxCreateString`, `mxCreateCharMatrixFromStrings`

Purpose

Copy COMPLEX*8 values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyComplex8ToPtr(y, pr, pi, n)
complex*8 y(n)
integer*4 pr, pi, n
```

Arguments

y
COMPLEX*8 Fortran array.

pr
Pointer to the real data of a single-precision MATLAB array.

pi
Pointer to the imaginary data of a single-precision MATLAB array.

n
Number of elements to copy.

Description

mxCopyComplex8ToPtr copies n COMPLEX*8 values from the Fortran COMPLEX*8 array y into the MATLAB arrays pointed to by pr and pi. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyPtrToComplex8, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyComplex16ToPtr

Purpose

Copy `COMPLEX*16` values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyComplex16ToPtr(y, pr, pi, n)
complex*16 y(n)
integer*4 pr, pi, n
```

Arguments

`y`
`COMPLEX*16` Fortran array.

`pr`
Pointer to the real data of a double-precision MATLAB array.

`pi`
Pointer to the imaginary data of a double-precision MATLAB array.

`n`
Number of elements to copy.

Description

`mxCopyComplex16ToPtr` copies `n` `COMPLEX*16` values from the Fortran `COMPLEX*16` array `y` into the MATLAB arrays pointed to by `pr` and `pi`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyPtrToComplex16`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose

Copy INTEGER*1 values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyInteger1ToPtr(y, px, n)
integer*1 y(n)
integer*4 px, n
```

Arguments

y
INTEGER*1 Fortran array.

px
Pointer to ir or jc array.

n
Number of elements to copy.

Description

mxCopyInteger1ToPtr copies n INTEGER*1 values from the Fortran INTEGER*1 array y into the MATLAB array pointed to by px, either an ir or jc array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Note This function can only be used with sparse matrices.

See Also

mxCopyPtrToInteger1, mxCreateNumericArray, mxCreateNumericMatrix

mxCopyInteger2ToPtr

Purpose

Copy INTEGER*2 values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyInteger2ToPtr(y, px, n)
integer*2 y(n)
integer*4 px, n
```

Arguments

y
INTEGER*2 Fortran array.

px
Pointer to ir or jc array.

n
Number of elements to copy.

Description

mxCopyInteger2ToPtr copies n INTEGER*2 values from the Fortran INTEGER*2 array y into the MATLAB array pointed to by px, either an ir or jc array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Note This function can only be used with sparse matrices.

See Also

mxCopyPtrToInteger2, mxCreateNumericArray, mxCreateNumericMatrix

Purpose Copy `INTEGER*4` values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyInteger4ToPtr(y, px, n)
integer*4 y(n)
integer*4 px, n
```

Arguments

`y`
`INTEGER*4` Fortran array.

`px`
Pointer to `ir` or `jc` array.

`n`
Number of elements to copy.

Description

`mxCopyInteger4ToPtr` copies `n` `INTEGER*4` values from the Fortran `INTEGER*4` array `y` into the MATLAB array pointed to by `px`, either an `ir` or `jc` array. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Note This function can only be used with sparse matrices.

See Also `mxCopyPtrToInteger4`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

mxCopyPtrToCharacter

Purpose

Copy character values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToCharacter(px, y, n)
character*(*) y
integer*4 px, n
```

Arguments

px
Pointer to character or name array.

y
character Fortran array.

n
Number of elements to copy.

Description

mxCopyPtrToCharacter copies n character values from the MATLAB array pointed to by px into the Fortran character array y. This subroutine is essential for copying character data from MATLAB pointer arrays into ordinary Fortran character arrays.

Example

See matdemo2.f in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

mxCopyCharacterToPtr, mxCreateCharArray, mxCreateString, mxCreateCharMatrixFromStrings

Purpose Copy COMPLEX*8 values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToComplex8(pr, pi, y, n)
complex*8 y(n)
integer*4 pr, pi, n
```

Arguments

pr
Pointer to the real data of a single-precision MATLAB array.

pi
Pointer to the imaginary data of a single-precision MATLAB array.

y
COMPLEX*8 Fortran array.

n
Number of elements to copy.

Description mxCopyPtrToComplex8 copies n COMPLEX*8 values from the MATLAB arrays pointed to by pr and pi into the Fortran COMPLEX*8 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also mxCopyComplex8ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyPtrToComplex16

Purpose

Copy `COMPLEX*16` values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToComplex16(pr, pi, y, n)
complex*16 y(n)
integer*4 pr, pi, n
```

Arguments

`pr`
Pointer to the real data of a double-precision MATLAB array.

`pi`
Pointer to the imaginary data of a double-precision MATLAB array.

`y`
`COMPLEX*16` Fortran array.

`n`
Number of elements to copy.

Description

`mxCopyPtrToComplex16` copies `n` `COMPLEX*16` values from the MATLAB arrays pointed to by `pr` and `pi` into the Fortran `COMPLEX*16` array `y`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyComplex16ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose Copy INTEGER*1 values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToInteger1(px, y, n)
integer*1 y(n)
integer*4 px, n
```

Arguments

`px`
Pointer to `ir` or `jc` array.

`y`
`INTEGER*1` Fortran array.

`n`
Number of elements to copy.

Description

`mxCopyPtrToInteger1` copies `n` `INTEGER*1` values from the MATLAB array pointed to by `px`, either an `ir` or `jc` array, into the Fortran `INTEGER*1` array `y`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Note This function can only be used with sparse matrices.

See Also

`mxCopyInteger1ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

mxCopyPtrToInteger2

Purpose

Copy INTEGER*2 values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToInteger2(px, y, n)
integer*2 y(n)
integer*4 px, n
```

Arguments

px
Pointer to ir or jc array.

y
INTEGER*2 Fortran array.

n
Number of elements to copy.

Description

mxCopyPtrToInteger2 copies n INTEGER*2 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER*2 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Note This function can only be used with sparse matrices.

See Also

mxCopyInteger2ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

Purpose Copy INTEGER*4 values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToInteger4(px, y, n)
integer*4 y(n)
integer*4 px, n
```

Arguments

px
Pointer to ir or jc array.

y
INTEGER*4 Fortran array.

n
Number of elements to copy.

Description mxCopyPtrToInteger4 copies n INTEGER*4 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER*4 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Note This function can only be used with sparse matrices.

See Also mxCopyInteger4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

mxCopyPtrToPtrArray

Purpose Copy pointer values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToPtrArray(px, y, n)
integer*4 y(n)
integer*4 px, n
```

Arguments

`px`
Pointer to pointer array.

`y`
`INTEGER*4` Fortran array.

`n`
Number of pointers to copy.

Description `mxCopyPtrToPtrArray` copies `n` pointers from the MATLAB array pointed to by `px` into the Fortran array `y`. This subroutine is essential for copying the output of `matGetDir` into an array of pointers. After calling this function, each element of `y` contains a pointer to a string. You can convert these strings to Fortran character arrays by passing each element of `y` as the first argument to `mxCopyPtrToCharacter`.

Example See `matdemo2.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also `matGetDir`, `mxCopyPtrToCharacter`

Purpose

Copy REAL*4 values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToReal4(px, y, n)
real*4 y(n)
integer*4 px, n
```

Arguments

px
Pointer to the real or imaginary data of a single-precision MATLAB array.

y
REAL*4 Fortran array.

n
Number of elements to copy.

Description

mxCopyPtrToReal4 copies n REAL*4 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*4 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyReal4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyPtrToReal8

Purpose

Copy `REAL*8` values from a pointer array to a Fortran array

Fortran Syntax

```
subroutine mxCopyPtrToReal8(px, y, n)
real*8 y(n)
integer*4 px, n
```

Arguments

`px`
Pointer to the real or imaginary data of a double-precision MATLAB array.

`y`
`REAL*8` Fortran array.

`n`
Number of elements to copy.

Description

`mxCopyPtrToReal8` copies `n` `REAL*8` values from the MATLAB array pointed to by `px`, either a `pr` or `pi` array, into the Fortran `REAL*8` array `y`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Example

See `fengdemo.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

`mxCopyReal8ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose

Copy REAL*4 values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyReal4ToPtr(y, px, n)
real*4 y(n)
integer*4 px, n
```

Arguments

y
REAL*4 Fortran array.

px
Pointer to the real or imaginary data of a single-precision MATLAB array.

n
Number of elements to copy.

Description

mxCopyReal4ToPtr(y,px,n) copies n REAL*4 values from the Fortran REAL*4 array y into the MATLAB array pointed to by px, either a pr or pi array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyPtrToReal4, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyReal8ToPtr

Purpose Copy REAL*8 values from a Fortran array to a pointer array

Fortran Syntax

```
subroutine mxCopyReal8ToPtr(y, px, n)
real*8 y(n)
integer*4 px, n
```

Arguments

y
REAL*8 Fortran array.

px
Pointer to the real or imaginary data of a double-precision MATLAB array.

n
Number of elements to copy.

Description `mxCopyReal8ToPtr(y,px,n)` copies `n` REAL*8 values from the Fortran REAL*8 array `y` into the MATLAB array pointed to by `px`, either a `pr` or `pi` array. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Example See `matdemo1.f` and `fengdemo.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also `mxCopyPtrToReal8`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose Create an unpopulated N-dimensional cell mxArray

Fortran Syntax

```
integer*4 function mxCreateCellArray(ndim, dims)
integer*4 ndim, dims
```

Arguments

ndim
The desired number of dimensions in the created cell. For example, to create a three-dimensional cell mxArray, set ndim to 3.

dims
The dimensions array. Each element in the dimensions array contains the size of the mxArray in that dimension. For example, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 mxArray. In most cases, there should be ndim elements in the dims array.

Returns A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellArray returns 0. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. The most common cause of failure is insufficient free heap space.

Description Use mxCreateCellArray to create a cell mxArray whose size is defined by ndim and dims. For example, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set

```
ndim = 3;
dims(1) = 4; dims(2) = 8; dims(3) = 7;
```

The created cell mxArray is unpopulated; that is, mxCreateCellArray initializes each cell to 0. To put data into a cell, call mxSetCell.

See Also mxCreateCellMatrix, mxGetCell, mxSetCell, mxIsCell

mxCreateCellMatrix

Purpose

Create an unpopulated two-dimensional cell `mxArray`

Fortran Syntax

```
integer*4 function mxCreateCellMatrix(m, n)
integer*4 m, n
```

Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

Returns

A pointer to the created cell `mxArray`, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateCellMatrix` returns `0`. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for `mxCreateCellMatrix` to be unsuccessful.

Description

Use `mxCreateCellMatrix` to create an `m`-by-`n` two-dimensional cell `mxArray`. The created cell `mxArray` is unpopulated; that is, `mxCreateCellMatrix` initializes each cell to `0`. To put data into the cells, call `mxSetCell`.

`mxCreateCellMatrix` is identical to `mxCreateCellArray` except that `mxCreateCellMatrix` can create two-dimensional `mxArrays` only, but `mxCreateCellArray` can create `mxArrays` having any number of dimensions greater than 1.

See Also

`mxCreateCellArray`

Purpose

Create an unpopulated N-dimensional character mxArray

Fortran Syntax

```
integer*4 function mxCreateCharArray(ndim, dims)
integer*4 ndim, dims
```

Arguments

ndim
The desired number of dimensions in the character mxArray. You must specify a positive number. If you specify 0, 1, or 2, mxCreateCharArray creates a two-dimensional mxArray.

dims
The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 character mxArray. The dims array must have at least ndim elements.

Returns

A pointer to the created character mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCharArray returns 0. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for mxCreateCharArray to be unsuccessful.

Description

Use mxCreateCharArray to create an mxArray of characters whose size is defined by ndim and dims. For example, to establish a two-dimensional mxArray of characters having dimensions 12-by-3, set

```
ndim = 2;
dims(1) = 12; dims(2) = 3;
```

The created mxArray is unpopulated; that is, mxCreateCharArray initializes each character to INTEGER*2 0.

See Also

mxCreateString

mxCreateCharMatrixFromStrings

Purpose

Create a populated two-dimensional char `mxArray`

Fortran Syntax

```
integer*4 function mxCreateCharMatrixFromStrings(m, str)
integer*4 m
character*(*) str(m)
```

Arguments

`m`
The desired number of rows in the created string `mxArray`. The value you specify for `m` should equal the size of the `str` array.

`str`
A Fortran `character*n` array of size `m`, where each element of the array is `n` bytes.

Returns

A pointer to the created char `mxArray`, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateCharMatrixFromStrings` returns `0`. If unsuccessful in a MEX-file, the MEX-file terminates, and control returns to the MATLAB prompt. Insufficient free heap space is the primary reason for `mxCreateCharMatrixFromStrings` to be unsuccessful. Another possible reason for failure is that `str` contains fewer than `m` strings.

Description

Use `mxCreateCharMatrixFromStrings` to create a two-dimensional string `mxArray`, where each row is initialized to `str`. The created `mxArray` has dimensions `m`-by-`n`, where `n` is the length of the number of characters in `str(i)`.

See Also

`mxCreateCharArray`, `mxCreateString`

Purpose

Create an unpopulated two-dimensional, double-precision, floating-point mxArray

Fortran Syntax

```
integer*4 function mxCreateDoubleMatrix(m, n, ComplexFlag)
integer*4 m, n, ComplexFlag
```

Arguments

m
The desired number of rows.

n
The desired number of columns.

ComplexFlag
If the data you plan to put into the mxArray has no imaginary component, specify 0. If the data has some imaginary components, specify 1.

Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateDoubleMatrix returns 0. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateDoubleMatrix is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Use mxCreateDoubleMatrix to create an m-by-n mxArray.

If you set ComplexFlag to 0, mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements and initializes each element to 0.0.

If you set ComplexFlag to 1, mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements and m-by-n imaginary elements. It initializes each real and imaginary element to 0.0.

Call mxDestroyArray when you finish using the mxArray. mxDestroyArray deallocates the mxArray and its associated real and complex elements.

See Also

mxCreateNumericArray

mxCreateFull (Obsolete)

Purpose

Create an unpopulated two-dimensional `mxArray`

Description

This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxCreateDoubleMatrix` instead.

See Also

`mxCreateSparse`

Purpose

Create an unpopulated N-dimensional numeric mxArray

Fortran Syntax

```
integer*4 function mxCreateNumericArray(ndim, dims, classid,
   ComplexFlag)
integer*4 ndim, dims, classid, ComplexFlag
```

Arguments

ndim
Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateNumericArray automatically sets the number of dimensions to 2.

dims
The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 mxArray. In most cases, there should be ndim elements in the dims array.

classid
A numerical identifier that represents a particular MATLAB class. Use the function, mxClassIDFromClassName, to derive the classid value from a class name character array.

The classid tells MATLAB how you want the numerical array data to be represented in memory. For example, specifying the int32 class causes each piece of numerical data in the mxArray to be represented as a 32-bit signed integer.

mxCreateNumericArray accepts any of the MATLAB signed numeric classes, shown to the left in the table below.

ComplexFlag
If the data you plan to put into the mxArray has no imaginary components, specify 0. If the data will have some imaginary components, specify 1.

Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateNumericArray returns 0. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateNumericArray is unsuccessful when there is not enough free heap space to create the mxArray.

mxCreateNumericArray

Description

Call `mxCreateNumericArray` to create an N-dimensional `mxArray` in which all data elements have the numeric data type specified by `classid`. After creating the `mxArray`, `mxCreateNumericArray` initializes all its real data elements to 0. If `ComplexFlag` is set to `1`, `mxCreateNumericArray` also initializes all its imaginary data elements to 0.

The following table shows the Fortran data types that are equivalent to MATLAB classes. Use these as shown in the example below.

MATLAB Class Name	Fortran Type
`int8`	`INTEGER*1`
`int16`	`INTEGER*2`
`int32`	`INTEGER*4`
`single`	`REAL*4`
`double`	`REAL*8`
`single`, with imaginary components	`COMPLEX*8`
`double`, with imaginary components	`COMPLEX*16`

`mxCreateNumericArray` differs from `mxCreateDoubleMatrix` in two important respects:

- All data elements in `mxCreateDoubleMatrix` are double-precision, floating-point numbers. The data elements in `mxCreateNumericArray` could be any numerical type, including different integer precisions.
- `mxCreateDoubleMatrix` can create two-dimensional arrays only; `mxCreateNumericArray` can create arrays of two or more dimensions.

`mxCreateNumericArray` allocates dynamic memory to store the created `mxArray`. When you finish with the created `mxArray`, call `mxDestroyArray` to deallocate its memory.

Example

To create a 4-by-4-by-2 array of REAL*8 elements having no imaginary components, use

```
C      Create 4x4x2 mxArray of REAL*8
       data dims / 4, 4, 2 /
       mxCreateNumericArray(3, dims,
     +               mxClassIDFromClassName('double'), 0)
```

See Also

mxCreateDoubleMatrix, mxCreateNumericMatrix, mxCreateSparse, mxCreateString

mxCreateNumericMatrix

Purpose

Create a numeric matrix and initialize all its data elements to 0

Fortran Syntax

```
integer*4 function mxCreateNumericMatrix(m, n, classid,
   ComplexFlag)
integer*4 m, n, classid, ComplexFlag
```

Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

`classid`
A numerical identifier that represents a particular MATLAB class. Use the function, `mxClassIDFromClassName`, to derive the `classid` value from a class name `character` array.

The `classid` tells MATLAB how you want the numerical array data to be represented in memory. For example, specifying the `int32` class causes each piece of numerical data in the `mxArray` to be represented as a 32-bit signed integer.

`mxCreateNumericMatrix` accepts any of the MATLAB signed numeric classes, shown to the left in the table below.

`ComplexFlag`
If the data you plan to put into the `mxArray` has no imaginary components, specify `0`. If the data has some imaginary components, specify `1`.

Returns

A pointer to the created `mxArray`, if successful. `mxCreateNumericMatrix` is unsuccessful if there is not enough free heap space to create the `mxArray`. If `mxCreateNumericMatrix` is unsuccessful in a MEX-file, the MEX-file prints an `Out of Memory` message, terminates, and control returns to the MATLAB prompt. If `mxCreateNumericMatrix` is unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateNumericMatrix` returns `0`.

Description

Call `mxCreateNumericMatrix` to create an two-dimensional `mxArray` in which all data elements have the numeric data type specified by `classid`. After creating the `mxArray`, `mxCreateNumericMatrix` initializes all its real data elements to 0. If `ComplexFlag` is set to `1`, `mxCreateNumericMatrix` also initializes all its imaginary data elements to 0. `mxCreateNumericMatrix` allocates dynamic memory to store the created `mxArray`. When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

The following table shows the Fortran data types that are equivalent to MATLAB classes. Use these as shown in the example below.

MATLAB Class Name	Fortran Type
`int8`	`BYTE`
`int16`	`INTEGER*2`
`int32`	`INTEGER*4`
`single`	`REAL*4`
`double`	`REAL*8`
`single`, with imaginary components	`COMPLEX*8`
`double`, with imaginary components	`COMPLEX*16`

Example

To create a 4-by-3 matrix of `REAL*4` elements having no imaginary components, use

```
C      Create 4x3 mxArray of REAL*4
       mxCreateNumericMatrix(4, 3,
     +                 mxClassIDFromClassName('single'), 0)
```

See Also

`mxCreateDoubleMatrix`, `mxCreateNumericArray`

mxCreateScalarDouble

Purpose

Create a scalar, double-precision array initialized to the specified value

Fortran Syntax

```
integer*4 function mxCreateScalarDouble(value)
real*4 value
```

Arguments

`value`
The desired value to which you want to initialize the array.

Returns

A pointer to the created `mxArray`, if successful. `mxCreateScalarDouble` is unsuccessful if there is not enough free heap space to create the `mxArray`. If `mxCreateScalarDouble` is unsuccessful in a MEX-file, the MEX-file prints an `Out of Memory` message, terminates, and control returns to the MATLAB prompt. If `mxCreateScalarDouble` is unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateScalarDouble` returns `0`.

Description

Call `mxCreateScalarDouble` to create a scalar double `mxArray`. `mxCreateScalarDouble` is a convenience function that can be used in place of the following code.

```
pm = mxCreateDoubleMatrix(1, 1, 0)
mxCopyReal8ToPtr(value, mxGetPr(pm), 1)
```

When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

See Also

`mxGetPr`, `mxCreateDoubleMatrix`

Purpose

Create a two-dimensional unpopulated sparse `mxArray`

Fortran Syntax

```
integer*4 function mxCreateSparse(m, n, nzmax, ComplexFlag)
integer*4 m, n, nzmax, ComplexFlag
```

Arguments

`m`
The desired number of rows.

`n`
The desired number of columns.

`nzmax`
The number of elements that `mxCreateSparse` should allocate to hold the `pr`, `ir`, and, if `ComplexFlag = 1`, `pi` arrays. Set the value of `nzmax` to be greater than or equal to the number of nonzero elements you plan to put into the `mxArray`, but make sure that `nzmax` is less than or equal to `m*n`.

`ComplexFlag`
Specify `REAL = 0` if the data has no imaginary components; specify `COMPLEX = 1` if the data has some imaginary components.

Returns

An unpopulated, sparse `mxArray` if successful, and 0 otherwise.

Description

Call `mxCreateSparse` to create an unpopulated sparse `mxArray`. The returned sparse `mxArray` contains no sparse information and cannot be passed as an argument to any MATLAB sparse functions. In order to make the returned sparse `mxArray` useful, you must initialize the `pr`, `ir`, `jc`, and (if it exists) `pi` array.

`mxCreateSparse` allocates space for

- A `pr` array of length `nzmax`.
- A `pi` array of length `nzmax` (but only if `ComplexFlag` is `COMPLEX = 1`).
- An `ir` array of length `nzmax`.
- A `jc` array of length `n+1`.

When you finish using the sparse `mxArray`, call `mxDestroyArray` to reclaim all its heap space.

See Also

`mxDestroyArray`, `mxSetNzmax`, `mxSetPr`, `mxSetIr`, `mxSetJc`

mxCreateString

Purpose

Create a 1-by-n `character` array initialized to the specified string

Fortran Syntax

```
integer*4 function mxCreateString(str)
character*(*) str
```

Arguments

`str`
The string that is to serve as the `mxArray` s initial data.

Returns

A `character` array initialized to `str` if successful, and 0 otherwise.

Description

Use `mxCreateString` to create a `character` `mxArray` initialized to `str`. Many MATLAB functions (for example, `strcmp` and `upper`) require `character` `mxArray` inputs.

Free the `character` `mxArray` when you are finished using it. To free a `character` `mxArray`, call `mxDestroyArray`.

Example

See `matdemo1.f` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

Purpose

Create an unpopulated N-dimensional structure mxArray

Fortran Syntax

```
integer*4 function mxCreateStructArray(ndim, dims, nfields,
   fieldnames)
integer*4 ndim, dims, nfields
character*(*) fieldnames(nfields)
```

Arguments

`ndim`
Number of dimensions. If you set `ndim` to be less than 2, `mxCreateStructArray` creates a two-dimensional `mxArray`.

`dims`
The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting `dims[1]` to `5` and `dims[2]` to `7` establishes a 5-by-7 `mxArray`. Typically, the `dims` array should have `ndim` elements.

`nfields`
The desired number of fields in each element.

`fieldnames`
The desired list of field names.

Returns

A pointer to the created structure `mxArray` if successful, and zero otherwise. The most likely cause of failure is insufficient heap space to hold the returned `mxArray`.

Description

Call `mxCreateStructArray` to create an unpopulated structure `mxArray`. Each element of a structure `mxArray` contains the same number of fields (specified in `nfields`). Each field has a name; the list of names is specified in `fieldnames`.

Each field holds one `mxArray` pointer. `mxCreateStructArray` initializes each field to zero. Call `mxSetField` or `mxSetFieldByNumber` to place a non-zero `mxArray` pointer in a field.

When you finish using the returned structure `mxArray`, call `mxDestroyArray` to reclaim its space.

See Also

`mxDestroyArray`, `mxCreateStructMatrix`, `mxIsStruct`, `mxAddField`, `mxSetField`, `mxGetField`, `mxRemoveField`

mxCreateStructMatrix

Purpose Create an unpopulated two-dimensional structure `mxArray`

Fortran Syntax

```
integer*4 function mxCreateStructMatrix(m, n, nfields, fieldnames)
integer*4 m, n, nfields
character*(*) fieldnames(nfields)
```

Arguments

`m`
The desired number of rows. This must be a positive integer.

`n`
The desired number of columns. This must be a positive integer.

`nfields`
The desired number of fields in each element.

`fieldnames`
The desired list of field names.

Returns A pointer to the created structure `mxArray` if successful, and `0` otherwise. The most likely cause of failure is insufficient heap space to hold the returned `mxArray`.

Description `mxCreateStructMatrix` and `mxCreateStructArray` are almost identical. The only difference is that `mxCreateStructMatrix` can only create two-dimensional `mxArrays`, while `mxCreateStructArray` can create `mxArrays` having two or more dimensions.

See Also `mxCreateStructArray`, `mxIsStruct`, `mxAddField`, `mxSetField`, `mxGetField`, `mxRemoveField`

Purpose Free dynamic memory allocated by an `mxCreate` routine

Fortran Syntax

```
subroutine mxDestroyArray(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to the `mxArray` that you want to free.

Description

`mxDestroyArray` deallocates the memory occupied by the specified `mxArray`. `mxDestroyArray` not only deallocates the memory occupied by the `mxArray`'s characteristics fields (such as `m` and `n`), but also deallocates all the `mxArray`'s associated data arrays (such as `pr`, `pi`, `ir`, and/or `jc`). You should not call `mxDestroyArray` on an `mxArray` you are returning on the left-hand side.

See Also `mxCalloc`, `mxFree`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

mxDuplicateArray

Purpose

Make a deep copy of an array

Fortran Syntax

```
integer*4 function mxDuplicateArray(in)
integer*4 in
```

Arguments

`in`
Pointer to the `mxArray` that you want to copy.

Returns

Pointer to a copy of the array.

Description

`mxDuplicateArray` makes a deep copy of an array, and returns a pointer to the copy. A deep copy refers to a copy in which all levels of data are copied. For example, a deep copy of a cell array copies each cell, and the contents of the each cell (if any), and so on.

Purpose

Free dynamic memory allocated by `mxCalloc`

Fortran Syntax

```
subroutine mxFree(ptr)
integer*4 ptr
```

Arguments

`ptr`
Pointer to the beginning of any memory parcel allocated by `mxCalloc`.

Description

`mxFree` deallocates heap space. `mxFree` frees memory using the MATLAB memory management facility. This ensures correct memory management in error and abort (**Ctrl-C**) conditions.

`mxFree` works differently in MEX-files than in stand-alone MATLAB applications. With MEX-files, `mxFree` returns to the heap any memory allocated using `mxCalloc`. If you do not free memory with this command, MATLAB frees it automatically on return from the MEX-file. In stand-alone MATLAB applications, you have to explicitly free memory, and MATLAB memory management is not used.

In a MEX-file, your use of `mxFree` depends on whether the specified memory parcel is persistent or nonpersistent. By default, memory parcels created by `mxCalloc` are nonpersistent.

The MATLAB memory management facility automatically frees all nonpersistent memory whenever a MEX-file completes. Thus, even if you do not call `mxFree`, MATLAB takes care of freeing the memory for you. Nevertheless, it is a good programming practice to deallocate memory just as soon as you are through using it. Doing so generally makes the entire system run more efficiently.

When a MEX-file completes, the MATLAB memory management facility does not free persistent memory parcels. Therefore, the only way to free a persistent memory parcel is to call `mxFree`. Typically, MEX-files call `mexAtExit` to register a clean-up handler. Then, the clean-up handler calls `mxFree`.

See Also

`mxCalloc`, `mxDestroyArray`

mxFreeMatrix (Obsolete)

Purpose Free dynamic memory allocated by `mxCreateFull` and `mxCreateSparse`

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxDestroyArray` instead.

See Also `mxCalloc`, `mxFree`

Purpose Get a cell's contents

Fortran Syntax

```
integer*4 function mxGetCell(pm, index)
integer*4 pm, index
```

Arguments

`pm`
Pointer to a cell `mxArray`.

`index`
The number of elements in the cell `mxArray` between the first element and the desired one. See `mxCalcSingleSubscript` for details on calculating an index in a multidimensional cell array.

Returns A pointer to the `i`th cell `mxArray` if successful, and `0` otherwise. Causes of failure include:

- The indexed cell array element has not been populated.
- Specifying an array pointer, `pm`, that does not point to a cell `mxArray`.
- Specifying an `index` greater than the number of elements in the cell.
- Insufficient free heap space to hold the returned cell `mxArray`.

Description Call `mxGetCell` to get a pointer to the `mxArray` held in the indexed element of the cell `mxArray`.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

See Also `mxCreateCellArray`, `mxIsCell`, `mxSetCell`

mxGetClassID

Purpose

Get an mxArray's class identifier

Fortran Syntax

```
integer*4 function mxGetClassID(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns

A numeric identifier that represents the class (category) of the mxArray that pm points to.

Description

Use mxGetClassId to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding.

See Also

mxGetClassName

Purpose Get (as a character array) an mxArray's class

Fortran Syntax

```
character*(*) function mxGetClassName(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns The class (as a character array) of mxArray, pm.

Description Call mxGetClassName to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For example, if pm points to a sparse mxArray, then mxGetClassName returns sparse.

See Also mxGetClassID

mxGetData

Purpose

Get pointer to data

Fortran Syntax

```
integer*4 function mxGetData(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns

The address of the first element of the real data, on success. Returns `0` if there is no real data or if there is an error.

Description

Call `mxGetData` to get a pointer to the real data in the `mxArray` that `pm` points to. To copy values from the pointer to Fortran, use one of the `mxCopyPtrTo*` functions in the manner shown here.

```
C      Get the data in mxArray, pm
       mxCopyPtrToReal8(mxGetData(pm), data,
     +                    mxGetNumberOfElements(pm))
```

`mxGetData` is equivalent to using `mxGetPr`.

See Also

`mxGetImagData`, `mxSetData`, `mxSetImagData`, `mxCopyPtrToReal4`, `mxCopyPtrToReal8`, `mxGetPr`

Purpose

Get a pointer to the dimensions array

Fortran Syntax

```
integer*4 function mxGetDimensions(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns

A pointer to the first element in a dimension array. Each integer in the dimensions array represents the number of elements in a particular dimension.

Description

Use `mxGetDimensions` to determine how many elements are in each dimension of the `mxArray` that `pm` points to. Call `mxGetNumberOfDimensions` to get the number of dimensions in the `mxArray`.

`mxGetDimensions` returns a pointer to the dimension array. To copy the values to Fortran, use `mxCopyPtrToInteger4` in the manner shown here.

```
C      Get dimensions of mxArray, pm
       mxCopyPtrToInteger4(mxGetDimensions(pm), dims,
     +                     mxGetNumberOfDimensions(pm))
```

See Also

`mxGetNumberOfDimensions`

mxGetElementSize

Purpose Get the number of bytes required to store each data element

Fortran Syntax

```
integer*4 function mxGetElementSize(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns The number of bytes required to store one element of the specified `mxArray`, if successful. Returns `0` on failure. The primary reason for failure is that `pm` points to an `mxArray` having an unrecognized class. If `pm` points to a cell `mxArray` or a structure `mxArray`, then `mxGetElementSize` returns the size of a pointer (not the size of all the elements in each cell or structure field).

Description Call `mxGetElementSize` to determine the number of bytes in each data element of the `mxArray`. For example, if the class of an `mxArray` is `int16`, then the `mxArray` stores each data element as a 16-bit (2 byte) signed integer. Thus, `mxGetElementSize` returns `2`.

See Also `mxGetM`, `mxGetN`

Purpose Get value of eps

Fortran Syntax

```
real*8 function mxGetEps
```

Returns The value of the MATLAB eps variable.

Description Call mxGetEps to return the value of the MATLAB eps variable. This variable holds the distance from 1.0 to the next largest floating-point number. As such, it is a measure of floating-point accuracy. The MATLAB pinv and rank functions use eps as a default tolerance.

See Also mxGetInf, mxGetNaN

mxGetField

Purpose

Get a field value, given a field name and an index in a structure array

Fortran Syntax

```
integer*4 function mxGetField(pm, index, fieldname)
integer*4 pm, index
character*(*) fieldname
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`index`
The desired element. The first element of an `mxArray` has an `index` of `1`, the second element has an `index` of `2`, and the last element has an `index` of `N`, where `N` is the total number of elements in the structure `mxArray`.

`fieldname`
The name of the field whose value you want to extract.

Returns

A pointer to the `mxArray` in the specified field at the specified `fieldname`, on success. Returns zero if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include:

- Specifying a `pm` that does not point to a structure `mxArray`. To determine if `pm` points to a structure `mxArray`, call `mxIsStruct`.
- Specifying an out-of-range `index` to an element past the end of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than 10.
- Specifying a nonexistent `fieldname`. Call `mxGetFieldNameByNumber` to get existing `field names`.
- Insufficient heap space to hold the returned `mxArray`.

Description

Call `mxGetField` to get the value held in the specified element of the specified field.

`mxGetFieldByNumber` is similar to `mxGetField`. Both functions return the same value. The only difference is in the way you specify the field. `mxGetFieldByNumber` takes `fieldnumber` as its third argument, and `mxGetField` takes `fieldname` as its third argument.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

Calling

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where `index` is `1` if you have a one-by-one structure.

See Also

`mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

mxGetFieldByNumber

Purpose

Get a field value, given a field number and an index in a structure array

Fortran Syntax

```
integer*4 function mxGetFieldByNumber(pm, index, fieldnumber)
integer*4 pm, index, fieldnumber
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`index`
The desired element. The first element of an `mxArray` has an `index` of `1`, the second element has an `index` of `2`, and the last element has an `index` of `N`, where `N` is the total number of elements in the structure `mxArray`.

`fieldnumber`
The position of the field whose value you want to extract. The first field within each element has a field number of `1`, the second field has a field number of `2`, and so on. The last field has a field number of `N`, where `N` is the number of fields.

Returns

A pointer to the `mxArray` in the specified field for the desired element, on success. Returns zero if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include:

- Specifying a `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to is a structure `mxArray`.
- Specifying an index < `1` or > the number of elements in the array.
- Specifying a nonexistent field number. Call `mxGetFieldNumber` to determine the field number that corresponds to a given field name.

Description

Call `mxGetFieldByNumber` to get the value held in the specified `fieldnumber` at the indexed element.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

Calling

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where `index` is `1` if you have a one-by-one structure.

See Also

`mxGetField`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxSetField`, `mxSetFieldByNumber`

mxGetFieldNameByNumber

Purpose Get a field name, given a field number in a structure array

Fortran Syntax

```
character*(*) function mxGetFieldNameByNumber(pm, fieldnumber)
integer*4 pm, fieldnumber
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`fieldnumber`
The position of the desired field. For instance, to get the name of the first field, set `fieldnumber` to `1`; to get the name of the second field, set `fieldnumber` to `2`; and so on.

Returns

The `n`th field name, on success. Returns `0` on failure. Common causes of failure include:

- Specifying a `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to a structure `mxArray`.
- Specifying a value of `fieldnumber` greater than the number of fields in the structure `mxArray`. (Remember that `fieldnumber` `1` represents the first field, so `index N` represents the last field.)

Description

Call `mxGetFieldNameByNumber` to get the name of a field in the given structure `mxArray`. A typical use of `mxGetFieldNameByNumber` is to call it inside a loop to get the names of all the fields in a given `mxArray`.

Consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

The `fieldnumber` `1` represents the field name `name`; `fieldnumber` `2` represents field name `billing`; `fieldnumber` `3` represents field name `test`. A `fieldnumber` other than `1`, `2`, or `3` causes `mxGetFieldNameByNumber` to return `0`.

See Also

`mxGetField`, `mxIsStruct`, `mxSetField`

Purpose Get a field number, given a field name in a structure array

Fortran Syntax

```
integer*4 function mxGetFieldNumber(pm, fieldname)
integer*4 pm
character*(*) fieldname
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`fieldname`
The name of a field in the structure `mxArray`.

Returns

The field number of the specified `fieldname`, on success. The first field has a field number of `1`, the second field has a field number of `2`, and so on. Returns `0` on failure. Common causes of failure include:

- Specifying a `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to a structure `mxArray`.
- Specifying the `fieldname` of a nonexistent field.

Description

If you know the name of a field but do not know its field number, call `mxGetFieldNumber`. Conversely, if you know the field number but do not know its field name, call `mxGetFieldNameByNumber`.

For example, consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

The field name `name` has a field number of `1`; the field name `billing` has a field number of `2`; and the field name `test` has a field number of `3`. If you call `mxGetFieldNumber` and specify a `fieldname` of anything other than `'name'`, `'billing'`, or `'test'`, then `mxGetFieldNumber` returns `0`.

Calling

```
mxGetField(pm, index, 'fieldname');
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname');
mxGetFieldByNumber(pm, index, fieldnum);
```

where `index` is `1` if you have a `1-by-1` structure.

See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxSetField`, `mxSetFieldByNumber`

Purpose Get pointer to imaginary data of an `mxArray`

Fortran Syntax

```
integer*4 function mxGetImagData(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns The address of the first element of the imaginary data, on success. Returns `0` if there is no imaginary data or if there is an error.

Description Call `mxGetImagData` to determine the starting address of the imaginary data in the `mxArray` that `pm` points to. To copy values from the pointer to Fortran, use one of the `mxCopyPtrToComplex*` functions in the manner shown here.

```
C      Get the real and imaginary data in mxArray, pm
       mxCopyPtrToComplex16(mxGetData(pm), mxGetImagData(pm),
     +                data, mxGetNumberOfElements(pm))
```

`mxGetImagData` is equivalent to using `mxGetPi`.

See Also `mxGetData`, `mxSetImagData`, `mxSetData`, `mxCopyPtrToComplex8`, `mxCopyPtrToComplex16`, `mxGetPi`

mxGetInf

Purpose Get the value of infinity

Fortran Syntax `real*8 function mxGetInf`

Returns The value of infinity on your system.

Description Call `mxGetInf` to return the value of the MATLAB internal `inf` variable. `inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `inf` is built into the system. You cannot modify it.

Operations that return infinity include:

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

See Also `mxGetEps`, `mxGetNaN`

Purpose Get the ir array

Fortran Syntax

```
integer*4 function mxGetIr(pm)
integer*4 pm
```

Arguments

pm
Pointer to a sparse mxArray.

Returns A pointer to the first element in the ir array if successful, and zero otherwise. Possible causes of failure include:

- Specifying a full (nonsparse) mxArray.
- An earlier call to mxCreateSparse failed.

Description Use mxGetIr to obtain the starting address of the ir array. The ir array is an array of integers; the length of the ir array is typically nzmax values. For example, if nzmax equals 100, then the ir array should contain 100 integers.

Each value in an ir array indicates a row (offset by 1) at which a nonzero element can be found. (The jc array is an index that indirectly specifies a column where nonzero elements can be found.)

For details on the ir and jc arrays, see mxSetIr and mxSetJc.

See Also mxGetJc, mxGetNzmax, mxSetIr, mxSetJc, mxSetNzmax

mxGetJc

Purpose

Get the jc array

Fortran Syntax

```
integer*4 function mxGetJc(pm)
integer*4 pm
```

Arguments

pm
Pointer to a sparse mxArray.

Returns

A pointer to the first element in the jc array if successful, and zero otherwise. The most likely cause of failure is specifying a pointer that points to a full (nonsparse) mxArray.

Description

Use mxGetJc to obtain the starting address of the jc array. The jc array is an integer array having n+1 elements where n is the number of columns in the sparse mxArray. The values in the jc array indirectly indicate columns containing nonzero elements. For a detailed explanation of the jc array, see mxSetJc.

See Also

mxGetIr, mxSetIr, mxSetJc

Purpose Get the number of rows

Fortran Syntax

```
integer*4 function mxGetM(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns The number of rows in the mxArray to which pm points.

Description mxGetM returns the number of rows in the specified array.

Example See matdemo2.f in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also mxGetN, mxSetM, mxSetN

mxGetN

Purpose Get the total number of columns

Fortran Syntax

```
integer*4 function mxGetN(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns The number of columns in the mxArray.

Description Call mxGetN to determine the number of columns in the specified mxArray.

If pm points to a sparse mxArray, mxGetN still returns the number of columns, not the number of occupied columns.

Example See matdemo2.f in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also mxGetM, mxSetM, mxSetN

Purpose

Get the name of the specified `mxArray`

Description

This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

mxGetNaN

Purpose Get the value of `NaN` (Not-a-Number)

Fortran Syntax

```
real*8 function mxGetNaN
```

Returns The value of `NaN` (Not-a-Number) on your system.

Description Call `mxGetNaN` to return the value of `NaN` for your system. `NaN` is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return `NaN` as a result, for example:

- `0.0/0.0`
- `Inf-Inf`

The value of Not-a-Number is built in to the system. You cannot modify it.

See Also `mxGetEps`, `mxGetInf`

Purpose Get the number of dimensions

Fortran Syntax

```
integer*4 function mxGetNumberOfDimensions(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns The number of dimensions in the specified `mxArray`. The returned value is always 2 or greater.

Description Use `mxGetNumberOfDimensions` to determine how many dimensions are in the specified array. To determine how many elements are in each dimension, call `mxGetDimensions`.

See Also `mxSetM`, `mxSetN`, `mxGetDimensions`

mxGetNumberOfElements

Purpose

Get number of elements in an array

Fortran Syntax

```
integer*4 function mxGetNumberOfElements(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns

Number of elements in the specified `mxArray`.

Description

`mxGetNumberOfElements` tells you how many elements an `mxArray` has. For example, if the dimensions of an array are 3-by-5-by-10, then `mxGetNumberOfElements` will return the number `150`.

See Also

`mxGetDimensions`, `mxGetM`, `mxGetN`, `mxGetClassName`

Purpose
Get the number of fields in a structure `mxArray`

Fortran Syntax

```
integer*4 function mxGetNumberOfFields(pm)
integer*4 pm
```

Arguments
`pm`
Pointer to a structure `mxArray`.

Returns
The number of fields, on success. Returns `0` on failure of if no fields exist. The most common cause of failure is that `pm` is not a structure `mxArray`. Call `mxIsStruct` to determine if `pm` is a structure.

Description
Call `mxGetNumberOfFields` to determine how many fields are in the specified structure `mxArray`.

Once you know the number of fields in a structure, it is easy to loop through every field to set or to get field values.

See Also
`mxGetField`, `mxIsStruct`, `mxSetField`

mxGetNzmax

Purpose

Get the number of elements in the `ir`, `pr`, and (if it exists) `pi` arrays

Fortran Syntax

```
integer*4 function mxGetNzmax(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to a sparse `mxArray`.

Returns

The number of elements allocated to hold nonzero entries in the specified sparse `mxArray`, on success. Returns an indeterminate value on error. The most likely cause of failure is that `pm` points to a full (nonsparse) `mxArray`.

Description

Use `mxGetNzmax` to get the value of the `nzmax` field. The `nzmax` field holds an integer value that signifies the number of elements in the `ir`, `pr`, and, if it exists, the `pi` arrays. The value of `nzmax` is always greater than or equal to the number of nonzero elements in a sparse `mxArray`. In addition, the value of `nzmax` is always less than or equal to the number of rows times the number of columns.

As you adjust the number of nonzero elements in a sparse `mxArray`, MATLAB often adjusts the value of the `nzmax` field. MATLAB adjusts `nzmax` in order to reduce the number of costly reallocations and in order to optimize its use of heap space.

See Also

`mxSetNzmax`

Purpose Get an mxArray's imaginary data elements

Fortran Syntax

```
integer*4 function mxGetPi(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns The imaginary data elements of the specified mxArray, on success. Returns 0 if there is no imaginary data or if there is an error.

Description Use mxGetPi to determine the starting address of the imaginary data in the mxArray that pm points to.

See the description for mxGetImagData, which is an equivalent function to mxGetPi.

See Also mxGetPr, mxSetPi, mxSetPr, mxGetImagData

mxGetPr

Purpose Get an mxArray's real data elements

Fortran Syntax

```
integer*4 function mxGetPr(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns The address of the first element of the real data. Returns 0 if there is no real data.

Description Use mxGetPr to determine the starting address of the real data in the mxArray that pm points to.

See the description for mxGetData, which is an equivalent function to mxGetPr.

Example See matdemo1.f and fengdemo.f in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also mxGetPi, mxSetPr, mxSetPi, mxGetData

Purpose Get the real component of an `mxArray`'s first data element

Fortran Syntax

```
real*8 function mxGetScalar(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns The value of the first real (nonimaginary) element of the `mxArray`. If `pm` points to a sparse `mxArray`, `mxGetScalar` returns the value of the first nonzero real element in the `mxArray`.

If `pm` points to an empty `mxArray`, `mxGetScalar` returns an indeterminate value.

Description Call `mxGetScalar` to get the value of the first real (nonimaginary) element of the `mxArray`.

In most cases, you call `mxGetScalar` when `pm` points to an `mxArray` containing only one element (a scalar). However, `pm` can point to an `mxArray` containing many elements. If `pm` points to an `mxArray` containing multiple elements, `mxGetScalar` returns the value of the first real element. If `pm` points to a two-dimensional `mxArray`, `mxGetScalar` returns the value of the `(1,1)` element.

See Also `mxGetM`, `mxGetN`

mxGetString

Purpose
Create a `character` array from an `mxArray`

Fortran Syntax

```
integer*4 function mxGetString(pm, str, strlen)
integer*4 pm, strlen
character*(*) str
```

Arguments

`pm`
Pointer to an `mxArray`.

`str`
Fortran `character` array.

`strlen`
Number of characters to retrieve from the `mxArray`.

Returns
0 on success, and 1 otherwise.

Description
Call `mxGetString` to copy a `character` array from an `mxArray`. `mxGetString` copies and converts the `character` array from the `mxArray` `pm` into the `character` array `str`. Storage space for `character` array `str` must be allocated previously.

Only up to `strlen` characters are copied, so ordinarily, `strlen` is set to the dimension of the `character` array to prevent writing past the end of the array. Check the length of the `character` array in advance using `mxGetM` and `mxGetN`. If the `character` array contains several rows, they are copied, one column at a time, into one long `character` array.

See Also
`mxCalloc`

Purpose True if a cell `mxArray`

Fortran Syntax

```
integer*4 function mxIsCell(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an array.

Returns `1` if `pm` points to an array of the MATLAB `cell` class, and `0` otherwise.

Description Use `mxIsCell` to determine if the specified `mxArray` is a cell array.

Calling `mxIsCell` is equivalent to calling

```
mxGetClassName(pm) .eq. 'cell'
```

Note `mxIsCell` does not answer the question, "Is this `mxArray` a cell of a cell array?". An individual cell of a cell array can be of any type.

See Also `mxIsClass`

mxIsChar

Purpose True if a character mxArray

Fortran Syntax

```
integer*4 function mxIsChar(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns 1 if pm points to an array of the MATLAB char class, and 0 otherwise.

Description Use mxIsChar to determine if the specified array is a character mxArray.

Calling mxIsChar is equivalent to calling

```
mxGetClassName(pm) .eq. 'char'
```

See Also mxIsClass, mxGetClassID

mxIsClass

Purpose

True if `mxArray` is a member of the specified class

Fortran Syntax

```
integer*4 function mxIsClass(pm, classname)
integer*4 pm
character*(*) classname
```

Arguments

`pm`
Pointer to an array.

`classname`
A `character` array specifying the class name you are testing for. You can specify any one of the following predefined constants.

`cell`	`char`	`double`	`function_handle`
`int8`	`int16`	`int32`	`object`
`single`	`sparse`	`struct`	`uint8`
`uint16`	`uint32`	`<class_name>`	`unknown`

In the table, `<class_name>` represents the name of a specific MATLAB custom object. You can also specify one of your own class names.

Returns

`1` if `pm` points to an array having category *`classname`*, and `0` otherwise.

Description

Each `mxArray` is tagged as being a certain type. Call `mxIsClass` to determine if the specified `mxArray` has this type.

Example

```
mxIsClass(pm, 'double')
```

is equivalent to calling either one of the following

```
mxIsDouble(pm)

mxGetClassName(pm) .eq. 'double'
```

It is more efficient to use the `mxIsDouble` form.

See Also

`mxIsEmpty`, `mxGetClassID`

mxIsComplex

Purpose Inquire if an `mxArray` is complex

Fortran Syntax

```
integer*4 function mxIsComplex(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns 1 if complex, and 0 otherwise.

Description Use `mxIsComplex` to determine whether or not an imaginary part is allocated for an `mxArray`. The imaginary pointer `pi` is 0 if an `mxArray` is purely real and does not have any imaginary data. If an `mxArray` is complex, `pi` points to an array of numbers.

See Also mxIsNumeric

Purpose

Inquire if an mxArray is of type double

Fortran Syntax

```
integer*4 function mxIsDouble(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns

1 if true, 0 if false. If mxIsDouble returns 0, the array has no Fortran access functions and your Fortran program cannot use it.

Description

Call mxIsDouble to determine whether or not the specified mxArray represents its real and imaginary data as double-precision, floating-point numbers.

Older versions of MATLAB store all mxArray data as double-precision, floating-point numbers. However, starting with MATLAB 5, MATLAB can store real and imaginary data in a variety of numerical formats.

Calling mxIsDouble is equivalent to calling

```
mxGetClassName(pm) .eq. 'double'
```

mxIsEmpty

Purpose True if mxArray is empty

Fortran Syntax

```
integer*4 function mxIsEmpty(pm)
integer*4 pm
```

Arguments

pm
Pointer to an array.

Returns 1 if the mxArray is empty, and 0 otherwise.

Description Use mxIsEmpty to determine if an mxArray contains no data. An mxArray is empty if the size of any of its dimensions is 0.

Note that mxIsEmpty is not the opposite of mxIsFull.

See Also mxIsClass

Purpose True if value is finite

Fortran Syntax

```
integer*4 function mxIsFinite(value)
real*8 value
```

Arguments

`value`
The double-precision, floating-point number that you are testing.

Returns `1` if `value` is finite, and `0` otherwise.

Description Call `mxIsFinite` to determine whether or not `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

See Also `mxIsInf`, `mxIsNaN`

mxIsFromGlobalWS

Purpose

True if the `mxArray` originated from the MATLAB global workspace

Fortran Syntax

```
integer*4 function mxIsFromGlobalWS(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns

`1` if the array originated from the global workspace, and `0` otherwise.

Description

Use `mxIsFromGlobalWS` with stand-alone MAT programs to determine if an array was a global variable when it was saved.

See Also

`mexIsGlobal`

Purpose Inquire if an `mxArray` is full

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use

```
if (mxIsSparse(prhs(1)) .eq. 0)
```

instead of

```
if (mxIsFull(prhs(1)) .eq. 1)
```

See Also `mxIsSparse`

mxIsInf

Purpose

True if value is infinite

Fortran Syntax

```
integer*4 function mxIsInf(value)
integer*4 value
```

Arguments

`value`
The double-precision, floating-point number that you are testing.

Returns

`1` if value is infinite, and `0` otherwise.

Description

Call `mxIsInf` to determine whether or not `value` is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable, `Inf`, is built into the system. You cannot modify it.

Operations that return infinity include:

- Division by 0. For example, `5/0` returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

See Also

`mxIsFinite`, `mxIsNaN`

Purpose True if mxArray represents its data as signed 8-bit integers

Fortran Syntax

```
integer*4 function mxIsInt8(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns 1 if the array stores its data as signed 8-bit integers, and 0 otherwise.

Description Use mxIsInt8 to determine whether or not the specified array represents its real and imaginary data as 8-bit signed integers.

Calling mxIsInt8 is equivalent to calling

```
mxGetClassName(pm) .eq. 'int8'
```

See Also mxIsClass, mxGetClassID

mxIsInt16

Purpose True if mxArray represents its data as signed 16-bit integers

Fortran Syntax

```
integer*4 function mxIsInt16(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns 1 if the array stores its data as signed 16-bit integers, and 0 otherwise.

Description Use mxIsInt16 to determine whether or not the specified array represents its real and imaginary data as 16-bit signed integers.

Calling mxIsInt16 is equivalent to calling

```
mxGetClassName(pm) == 'int16'
```

See Also mxIsClass, mxGetClassID

Purpose True if `mxArray` represents its data as signed 32-bit integers

Fortran Syntax

```
integer*4 function mxIsInt32(pm)
integer*4 pm
```

Arguments

`m`
Pointer to an `mxArray`.

Returns `1` if the array stores its data as signed 32-bit integers, and `0` otherwise.

Description Use `mxIsInt32` to determine whether or not the specified array represents its real and imaginary data as 32-bit signed integers.

Calling `mxIsInt32` is equivalent to calling

```
mxGetClassName(pm) == 'int32'
```

See Also `mxIsClass`, `mxGetClassID`

mxIsLogical

Purpose True if `mxArray` is Boolean

Fortran Syntax

```
integer*4 function mxIsLogical(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns `1` if the `mxArray`'s logical flag is on, and 0 otherwise. If an `mxArray` does not hold numeric data (for instance, if `pm` points to a structure `mxArray` or a cell `mxArray`), then `mxIsLogical` automatically returns `0`.

Description Use `mxIsLogical` to determine whether MATLAB treats the data in the `mxArray` as Boolean (logical) or numerical (not logical).

If an `mxArray` is logical, then MATLAB treats all zeros as meaning `false` and all nonzero values as meaning `true`. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

See Also `mxIsClass`, `mxSetLogical (Obsolete)`, `logical`

Purpose True if value is `NaN` (Not-a-Number)

Fortran Syntax

```
integer*4 function mxIsNaN(value)
integer*4 value
```

Arguments

`value`
The double-precision, floating-point number that you are testing.

Returns `1` if `value` is `NaN` (Not-a-Number), and `0` otherwise.

Description Call `mxIsNaN` to determine whether or not `value` is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. A `NaN` is obtained as a result of mathematically undefined operations such as:

- `0.0/0.0`
- `Inf-Inf`

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value, rather it is a family of numbers that MATLAB (and other IEEE-compliant applications) uses to represent an error condition or missing data.

See Also `mxIsFinite`, `mxIsInf`

mxIsNumeric

Purpose Inquire if an mxArray contains numeric data

Fortran Syntax

```
integer*4 function mxIsNumeric(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns 1 if the mxArray contains numeric data, and 0 otherwise.

Description Call mxIsNumeric to inquire whether or not the mxArray contains a character array.

Example See matdemo1.f in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also mxIsString (Obsolete)

Purpose
True if mxArray represents its data as single-precision, floating-point numbers

Fortran Syntax
```
integer*4 function mxIsSingle(pm)
integer*4 pm
```

Arguments
pm
Pointer to an mxArray.

Returns
1 if the array stores its data as single-precision, floating-point numbers, and 0 otherwise.

Description
Use mxIsSingle to determine whether or not the specified array represents its real and imaginary data as single-precision, floating-point numbers.

Calling mxIsSingle is equivalent to calling

```
mxGetClassName(pm) .eq. 'single'
```

See Also
mxIsClass, mxGetClassID

mxIsSparse

Purpose Inquire if an mxArray is sparse

Fortran Syntax

```
integer*4 function mxIsSparse(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns 1 if the mxArray is sparse, and 0 otherwise.

Description Use mxIsSparse to determine if an mxArray is stored in sparse form. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.

There are no corresponding set routines. Use mxCreateSparse to create sparse mxArrays.

See Also mxGetIr, mxGetJc, mxCreateSparse

Purpose Inquire if an `mxArray` contains a `character` array

Description This API function is obsolete and is not supported in MATLAB 6.1 or later. This function may not be available in a future version of MATLAB.

Use `mxIsChar` instead.

See Also `mxCreateString`, `mxGetString`

mxIsStruct

Purpose True if a structure `mxArray`

Fortran Syntax

```
integer*4 function mxIsStruct(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns `1` if `pm` points to a structure `array`; otherwise, `0`.

Description Use `mxIsStruct` to determine if `pm` points to a structure `mxArray`. Many routines (for example, `mxGetFieldName` and `mxSetField`) require a structure `mxArray` as an argument.

See Also `mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetNumberOfFields`, `mxGetField`, `mxSetField`

Purpose True if mxArray represents its data as unsigned 8-bit integers

Fortran Syntax

```
integer*4 function mxIsInt8(pm)
integer*4 pm
```

Arguments

m
Pointer to an mxArray.

Returns 1 if the mxArray stores its data as unsigned 8-bit integers, and 0 otherwise.

Description Use mxIsInt8 to determine whether or not the specified mxArray represents its real and imaginary data as 8-bit unsigned integers.

Calling mxIsUint8 is equivalent to calling

```
mxGetClassName(pm) == 'uint8'
```

See Also mxGetClassID, mxIsClass, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint16, mxIsUint32

mxIsUint16

Purpose
True if mxArray represents its data as unsigned 16-bit integers

Fortran Syntax

```
integer*4 function mxIsUint16(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns
1 if the mxArray stores its data as unsigned 16-bit integers, and 0 otherwise.

Description
Use mxIsUint16 to determine whether or not the specified mxArray represents its real and imaginary data as 16-bit unsigned integers.

Calling mxIsUint16 is equivalent to calling

```
mxGetClassName(pm) == 'uint16'
```

See Also
mxGetClassID, mxIsClass, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint8, mxIsUint32

Purpose True if mxArray represents its data as unsigned 32-bit integers

Fortran Syntax

```
integer*4 function mxIsUint32(pm)
integer*4 pm
```

Arguments

pm
Pointer to an mxArray.

Returns 1 if the mxArray stores its data as unsigned 32-bit integers, and 0 otherwise.

Description Use mxIsUint32 to determine whether or not the specified mxArray represents its real and imaginary data as 32-bit unsigned integers.

Calling mxIsUint32 is equivalent to calling

```
mxGetClassName(pm) == 'uint32'
```

See Also mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint8, mxIsUint16

mxMalloc

Purpose Allocate dynamic memory using the MATLAB memory manager

Fortran Syntax

```
integer*4 function mxMalloc(n)
integer*4 n
```

Arguments

`n`
Number of bytes to allocate.

Returns A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxMalloc` returns `0`. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.

`mxMalloc` is unsuccessful when there is insufficient free heap space.

Description Use `mxMalloc` to allocate dynamic memory using the MATLAB memory management facility.

MATLAB maintains a list of all memory allocated by `mxMalloc`. MATLAB automatically frees (deallocates) all of a MEX-file's memory when the MEX-file completes and control returns to the MATLAB prompt.

If you want the memory to persist after a MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxMalloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX-file is cleared.

When you finish using the memory allocated by `mxMalloc`, call `mxFree`. `mxFree` deallocates the memory.

Note that `mxMalloc` works differently in MEX-files than in stand-alone MATLAB applications.

In MEX-files, `mxMalloc` automatically:

- Allocates enough contiguous heap space to hold `n` bytes.
- Registers the returned heap space with the MATLAB memory management facility.

See Also `mxCalloc`, `mxFree`, `mxDestroyArray`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

mxRealloc

Purpose Reallocate memory

Fortran Syntax

```
integer*4 function mxRealloc(ptr, size)
integer*4 ptr, size
```

Arguments

`ptr`
Pointer to a block of memory allocated by `mxCalloc`, or by a previous call to `mxRealloc`.

`size`
New size of allocated memory, in bytes.

Returns A pointer to the reallocated block of memory on success, and `0` on failure.

Description `mxRealloc` reallocates the memory routine for the managed list. If `mxRealloc` fails to allocate a block, you must free the block since the ANSI definition of `realloc` states that the block remains allocated. `mxRealloc` returns `0` in this case, and in subsequent calls to `mxRealloc` of the form

```
x = mxRealloc(x, size)
```

Note Failure to reallocate memory with `mxRealloc` can result in memory leaks.

See Also `mxCalloc`, `mxFree`, `mxMalloc`

mxRemoveField

Purpose Remove a field from a structure array

Fortran Syntax

```
subroutine mxRemoveField(pm, fieldnumber)
integer*4 pm, fieldnumber
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`fieldnumber`
The number of the field you want to remove. For instance, to remove the first field, set `fieldnumber` to `1`; to remove the second field, set `fieldnumber` to `2`; and so on.

Description

Call `mxRemoveField` to remove a field from a structure array. If the field does not exist, nothing happens. This function does not destroy the field values. Use `mxDestroyArray` to destroy the actual field values.

Consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

The `fieldnumber` `1` represents the field name `name`; `fieldnumber` `2` represents field name `billing`; `fieldnumber` `3` represents field name `test`.

See Also

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

Purpose Set the value of one cell

Fortran Syntax

```
subroutine mxSetCell(pm, index, value)
integer*4 pm, index, value
```

Arguments

`pm`
Pointer to a cell `mxArray`.

`index`
Index from the beginning of the `mxArray`. Specify the number of elements between the first cell of the `mxArray` and the cell you want to set. The easiest way to calculate the `index` in a multidimensional cell array is to call `mxCalcSingleSubscript`.

`value`
The new value of the cell. You can put any kind of `mxArray` into a cell. In fact, you can even put another cell `mxArray` into a cell. Use one of the `mxCreate*` functions to create the `value` `mxArray`.

Description

Call `mxSetCell` to put the designated `value` into a particular cell of a cell `mxArray`. You can assign new values to unpopulated cells or overwrite the value of an existing cell. To do the latter, first use `mxDestroyArray` to free what is already there and then `mxSetCell` to assign the new value.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

See Also

`mxCreateCellArray`, `mxCreateCellMatrix`, `mxGetCell`, `mxIsCell`

mxSetData

Purpose

Set pointer to data

Fortran Syntax

```
subroutine mxSetData(pm, pr)
integer*4 pm, pr
```

Arguments

`pm`
Pointer to an `mxArray`.

`pr`
Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory.

Description

Use `mxSetData` to set the real data of the specified `mxArray`.

All `mxCreate*` calls allocate heap space to hold real data. Therefore, you do not ordinarily use `mxSetData` to initialize the real elements of a freshly created `mxArray`. Rather, you call `mxSetData` to replace the initial real values with new ones.

Free the memory used by `pr` by calling `mxDestroyArray` to destroy the entire `mxArray`.

`mxSetData` is equivalent to using `mxSetPr`.

See Also

`mxSetImagData`, `mxGetData`, `mxGetImagData`, `mxSetPr`

Purpose Modify the number of dimensions and/or the size of each dimension

Fortran Syntax

```
integer*4 function mxSetDimensions(pm, dims, ndim)
integer*4 pm, dims, ndim
```

Arguments

`pm`
Pointer to an `mxArray`.

`dims`
The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a 5-by-7 `mxArray`. In most cases, there should be `ndim` elements in the `dims` array.

`ndim`
The desired number of dimensions.

Returns

`0` on success, and `1` on failure. `mxSetDimensions` allocates heap space to hold the input size array. So it is possible (though extremely unlikely) that increasing the number of dimensions can cause the system to run out of heap space.

Description

Call `mxSetDimensions` to reshape an existing `mxArray`. `mxSetDimensions` is similar to `mxSetM` and `mxSetN`; however, `mxSetDimensions` provides greater control for reshaping `mxArrays` that have more than two-dimensions.

`mxSetDimensions` does not allocate or deallocate any space for the `pr` or `pi` array. Consequently, if your call to `mxSetDimensions` increases the number of elements in the `mxArray`, then you must enlarge the `pr` (and `pi`, if it exists) array accordingly.

If your call to `mxSetDimensions` reduces the number of elements in the `mxArray`, then you can optionally reduce the size of the `pr` and `pi` arrays using `mxRealloc`.

See Also

`mxGetNumberOfDimensions`, `mxSetM`, `mxSetN`

mxSetField

Purpose Set a field value of a structure array, given a field name and an index

Fortran Syntax

```
subroutine mxSetField(pm, index, fieldname, value)
integer*4 pm, index, value
character*(*) fieldname
```

Arguments

`pm`
Pointer to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to a structure `mxArray`.

`index`
The desired element. The first element of an `mxArray` has an index of `1`, the second element has an index of `2`, and the last element has an index of `N`, where `N` is the total number of elements in the structure `mxArray`.

`fieldname`
The name of the field whose value you are assigning. Call `mxGetFieldNameByNumber` to determine existing field names.

`value`
Pointer to the `mxArray` you are assigning. Use one of the `mxCreate*` functions to create the `value` `mxArray`.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

Description Use `mxSetField` to assign a `value` to the specified element of the specified field. If there is already a value at the given position, the `value` pointer you specified overwrites the old value pointer. However, `mxSetField` does not free the dynamic memory that the old value pointer pointed to. Consequently, you are responsible for destroying this `mxArray`.

`mxSetField` is almost identical to `mxSetFieldByNumber`; however, the former takes a field name as its third argument, and the latter takes a field number as its third argument.

Calling

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetFieldByNumber`

mxSetFieldByNumber

Purpose

Set a field value in a structure array, given a field number and an index

Fortran Syntax

```
subroutine mxSetFieldByNumber(pm, index, fieldnumber, value)
integer*4 pm, index, fieldnumber, value
```

Arguments

`pm`
Pointer to a structure `mxArray`. Call `mxIsStruct` to determine if `pm` points to a structure `mxArray`.

`index`
The desired element. The first element of an `mxArray` has an index of `1`, the second element has an index of `2`, and the last element has an index of `N`, where `N` is the total number of elements in the structure `mxArray`.

`fieldnumber`
The position of the field whose value you want to extract. The first field within each element has a `fieldnumber` of `1`, the second field has a `fieldnumber` of `2`, and so on. The last field has a `fieldnumber` of `N`, where `N` is the number of fields.

`value`
The value you are assigning. Use one of the `mxCreate*` functions to create the `value` `mxArray`.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

Description

Use `mxSetFieldByNumber` to assign a `value` to the specified element of the specified field. If there is already a value at the given position, the `value` pointer you specified overwrites the old value pointer. However, `mxSetFieldByNumber` does not free the dynamic memory that the old value pointer pointed to. Consequently, you are responsible for destroying this `mxArray`.

`mxSetFieldByNumber` is almost identical to `mxSetField`; however, the former takes a field number as its third argument, and the latter takes a field name as its third argument.

Calling

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`

mxSetImagData

Purpose Set imaginary data pointer for an mxArray

Fortran Syntax

```
subroutine mxSetImagData(pm, pi)
integer*4 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 Use mxSetImagData to set the imaginary data of the specified mxArray.

Most mxCreate* functions optionally allocate heap space to hold imaginary data. If you tell an mxCreate* function to allocate heap space (for example, by setting the ComplexFlag to COMPLEX = 1 or by setting pi to a nonzero value), then you do not ordinarily use mxSetImagData to initialize the created mxArray's imaginary elements. Rather, you call mxSetImagData to replace the initial imaginary values with new ones.

Free the memory used by pi by calling mxDestroyArray to destroy the entire mxArray.

mxSetImagData is equivalent to using mxSetPi.

See Also mxSetData, mxGetImagData, mxGetData, mxSetPi

Purpose

Set the `ir` array of a sparse `mxArray`

Fortran Syntax

```
subroutine mxSetIr(pm, ir)
integer*4 pm,ir
```

Arguments

`pm`
Pointer to a sparse `mxArray`.

`ir`
Pointer to the `ir` array. The `ir` array must be sorted in column-major order.

Description

Use `mxSetIr` to specify the `ir` array of a sparse `mxArray`. The `ir` array is an array of integers; the length of the `ir` array should equal the value of `nzmax`.

Each element in the `ir` array indicates a row (offset by 1) at which a nonzero element can be found. (The `jc` array is an index that indirectly specifies a column where nonzero elements can be found. See `mxSetJc` for more details on `jc`.)

The `ir` array must be in column-major order. That means that the `ir` array must define the row positions in column 1 (if any) first, then the row positions in column 2 (if any) second, and so on through column N. Within each column, row position 1 must appear prior to row position 2, and so on.

`mxSetIr` does not sort the `ir` array for you; you must specify an `ir` array that is already sorted.

See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetJc`

mxSetJc

Purpose Set the jc array of a sparse mxArray

Fortran Syntax

```
subroutine mxSetJc(pm, jc)
integer*4 pm, jc
```

Arguments

pm
Pointer to a sparse mxArray.

jc
Pointer to the jc array.

Description Use mxSetJc to specify a new jc array for a sparse mxArray. The jc array is an integer array having n+1 elements where n is the number of columns in the sparse mxArray.

See Also mxGetIr, mxGetJc, mxSetIr

Purpose

Set the logical flag

Note As of MATLAB version 6.5, `mxSetLogical` is obsolete. Support for `mxSetLogical` may be removed in a future version.

Fortran Syntax

```
subroutine mxSetLogical(pm)
integer*4 pm
```

Arguments

`pm`
Pointer to an `mxArray` having a numeric class.

Description

Use `mxSetLogical` to turn on an `mxArray`'s logical flag. This flag, when set, tells MATLAB that the array's data is to be treated as Boolean. If the logical flag is on, then MATLAB treats a 0 value as meaning `false` and a nonzero value as meaning `true`. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

See Also

`mxClearLogical (Obsolete)`, `mxIsLogical`, `logical`

mxSetM

Purpose

Set the number of rows

Fortran Syntax

```
subroutine mxSetM(pm, m)
integer*4 pm, m
```

Arguments

`pm`
Pointer to an `mxArray`.

`m`
The desired number of rows.

Description

Call `mxSetM` to set the number of rows in the specified `mxArray`. Call `mxSetN` to set the number of columns.

You can use `mxSetM` to change the shape of an existing `mxArray`. Note that `mxSetM` does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. Consequently, if your calls to `mxSetM` and `mxSetN` increase the number of elements in the `mxArray`, then you must enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays.

If your calls to `mxSetM` and `mxSetN` end up reducing the number of elements in the array, then you may want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays in order to use heap space more efficiently.

See Also

`mxGetM`, `mxGetN`, `mxSetN`

Purpose Set the number of columns

Fortran Syntax

```
subroutine mxSetN(pm, n)
integer*4 pm, n
```

Arguments

`pm`
Pointer to an `mxArray`.

`n`
The desired number of columns.

Description

Call `mxSetN` to set the number of columns in the specified `mxArray`. Call `mxSetM` to set the number of rows in the specified `mxArray`.

You typically use `mxSetN` to change the shape of an existing `mxArray`. Note that `mxSetN` does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. Consequently, if your calls to `mxSetN` and `mxSetM` increase the number of elements in the `mxArray`, then you must enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays.

If your calls to `mxSetM` and `mxSetN` end up reducing the number of elements in the `mxArray`, then you may want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

See Also `mxGetM`, `mxGetN`, `mxSetM`

mxSetName (Obsolete)

Purpose Set the name of an `mxArray`

Description This API function is obsolete and is not supported in MATLAB 6.5 or later. This function may not be available in a future version of MATLAB.

Purpose Set the storage space for nonzero elements

Fortran Syntax

```
subroutine mxSetNzmax(pm, nzmax)
integer*4 pm, nzmax
```

Arguments

`pm`
Pointer to a sparse `mxArray`.

`nzmax`
The number of elements that `mxCreateSparse` should allocate to hold the arrays pointed to by `ir`, `pr`, and `pi` (if it exists). Set `nzmax` greater than or equal to the number of nonzero elements in the `mxArray`, but set it to be less than or equal to the number of rows times the number of columns. If you specify an `nzmax` value of 0, `mxSetNzmax` sets the value of `nzmax` to 1.

Description

Use `mxSetNzmax` to assign a new value to the `nzmax` field of the specified sparse `mxArray`. The `nzmax` field holds the maximum possible number of nonzero elements in the sparse `mxArray`.

The number of elements in the `ir`, `pr`, and `pi` (if it exists) arrays must be equal to `nzmax`. Therefore, after calling `mxSetNzmax`, you must change the size of the `ir`, `pr`, and `pi` arrays.

How big should `nzmax` be? One thought is that you set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space. Another technique is to make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

See Also

`mxGetNzmax`

mxSetPi

Purpose

Set new imaginary data for an `mxArray`

Fortran Syntax

```
subroutine mxSetPi(pm, pi)
integer*4 pm, pi
```

Arguments

`pm`
Pointer to a full (nonsparse) `mxArray`.

`pi`
Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory. If `pi` points to static memory, memory errors will result when the array is destroyed.

Description

Use `mxSetPi` to set the imaginary data of the specified `mxArray`.

See the description for `mxSetImagData`, which is an equivalent function to `mxSetPi`.

See Also

`mxSetPr`, `mxGetPi`, `mxGetPr`, `mxSetImagData`

Purpose Set new real data for an `mxArray`

Fortran Syntax

```
subroutine mxSetPr(pm, pr)
integer*4 pm, pr
```

Arguments

`pm`
Pointer to a full (nonsparse) `mxArray`.

`pr`
Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory.

Description Use `mxSetPr` to set the real data of the specified `mxArray`.

See the description for `mxSetData`, which is an equivalent function to `mxSetPr`.

See Also `mxSetPi`, `mxGetPr`, `mxGetPi`, `mxSetData`

9

Java Interface Functions

`class`	Create object or return class of object
`import`	Add a package or class to the current Java import list
`isa`	Detect an object of a given class
`isjava`	Test whether an object is a Java object
`javaArray`	Constructs a Java array
`javachk`	Generate an error message based on Java feature support
`javaMethod`	Invokes a Java method
`javaObject`	Constructs a Java object
`methods`	Display method names
`methodsview`	Displays information on all methods implemented by a class
`usejava`	Determine if a Java feature is supported in MATLAB

class

Purpose

Create object or return class of object

Syntax

```
str = class(object)
obj = class(s,'class_name')
obj = class(s,'class_name',parent1,parent2...)
obj = class(struct([]),'class_name',parent1,parent2...)
```

Description

`str = class(object)` returns a string specifying the class of `object`.

The following table lists the object class names that may be returned. All except the last one are MATLAB classes.

`logical`	Logical array of `true` and `false` values
`char`	Characters array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array
`single`	Single-precision floating point number array
`double`	Double-precision floating point number array
`cell`	Cell array
`struct`	Structure array
`function handle`	Array of values for calling functions indirectly
`'`*`class_name`*`'`	Custom MATLAB object class or Java class

`obj = class(s,'class_name')` creates an object of MATLAB class `'class_name'` using structure `s` as a template. This syntax is valid only in a function named `class_name.m` in a directory named `@class_name` (where `'class_name'` is the same as the string passed into `class`).

`obj = class(s,'class_name',parent1,parent2,...)` creates an object of MATLAB class `'class_name'` that inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. Structure `s` is used as a template for the object.

`obj = class(struct([]),'class_name',parent1,parent2,...)` creates an object of MATLAB class `'class_name'` that inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. Specifying the empty structure, `struct([])`, as the first argument ensures that the object created contains no fields other than those that are inherited from the parent objects.

Examples

To return in `nameStr` the name of the class of Java object `j`

```
nameStr = class(j)
```

To create a user-defined MATLAB object of class `polynom`

```
p = class(p,'polynom')
```

See Also

`inferiorto`, `isa`, `superiorto`

The "MATLAB Classes and Objects" and the "Calling Java from MATLAB" chapters in *Programming and Data Types*.

import

Purpose

Add a package or class to the current Java import list for the MATLAB command environment or for the calling function

Syntax

```
import package_name.*
import class_name
import cls_or_pkg_name1 cls_or_pkg_name2...
import
L = import
```

Description

`import` *`package_name`*`.*` adds all the classes in *`package_name`* to the current import list. Note that *`package_name`* must be followed by `.*`.

`import` *`class_name`* adds a single class to the current import list. Note that *`class_name`* must be fully qualified (that is, it must include the package name).

`import` *`cls_or_pkg_name1 cls_or_pkg_name2`*`...` adds all named classes and packages to the current import list. Note that each class name must be fully qualified, and each package name must be followed by `.*`.

`import` with no input arguments displays the current import list, without adding to it.

`L = import` with no input arguments returns a cell array of strings containing the current import list, without adding to it.

`The import` command operates exclusively on the import list of the function from which it is invoked. When invoked at the command prompt, `import` uses the import list for the MATLAB command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```

This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

Remarks

The only reason for using `import` is to allow your code to refer to each imported class with the immediate class name only, rather than with the fully qualified class name. `import` is particularly useful in streamlining calls to constructors, where most references to Java classes occur.

Examples

This example shows importing and using the single class, `java.lang.String`, and two complete packages, `java.util` and `java.awt`.

```
import java.lang.String
import java.util.* java.awt.*
f = Frame;               % Create java.awt.Frame object
s = String('hello');     % Create java.lang.String object
methods Enumeration      % List java.util.Enumeration methods
```

See Also

`clear`

Purpose Detect an object of a given MATLAB class or Java class

Syntax

```
K = isa(obj,'class_name')
```

Description

`K = isa(obj,'class_name')` returns logical true (`1`) if `obj` is of class (or a subclass of) *`class_name`*, and logical false (`0`) otherwise.

The argument `obj` is a MATLAB object or a Java object. The argument *`class_name`* is the name of a MATLAB (predefined or user-defined) or a Java class. Predefined MATLAB classes include:

`logical`	Logical array of `true` and `false` values
`char`	Characters array
`numeric`	Integer or floating-point array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array
`single`	Single-precision floating-point array
`double`	Double-precision floating-point array
`cell`	Cell array
`struct`	Structure array
`function_handle`	Function Handle
`'class_name'`	Custom MATLAB object class or Java class

To check for a sparse array, use `issparse`. To check for a complex array, use `~isreal`.

Examples

```
isa(rand(3,4),'double')
ans =
    1
```

The following example creates an instance of the user-defined MATLAB class, named `polynom`. The `isa` function identifies the object as being of the `polynom` class.

```
polynom_obj = polynom([1 0 -2 -5]);
isa(polynom_obj, 'polynom')
ans =
     1
```

See Also

`class`, is*

isjava

Purpose Determine if item is a Java object

Syntax

```
tf = isjava(A)
```

Description

`tf = isjava(A)` returns logical true (`1`) if `A` is a Java object, and logical false (`0`) otherwise.

Examples

Create an instance of the Java Frame class and `isjava` indicates that it is a Java object.

```
frame = java.awt.Frame('Frame A');

isjava(frame)

ans =

     1
```

Note that, `isobject`, which tests for MATLAB objects, returns false (`0`).

```
isobject(frame)

ans =

     0
```

See Also

`isobject`, `javaArray`, `javaMethod`, `javaObject`, `isa`, `is*`

Purpose Constructs a Java array

Syntax

```
javaArray('package_name.class_name ,x1,...,xn)
```

Description

`javaArray('package_name.class_name ,x1,...,xn)` constructs an empty Java array capable of storing objects of Java class, `'class_name'`. The dimensions of the array are `x1` by `...` by `xn`. You must include the package name when specifying the class.

The array that you create with `javaArray` is equivalent to the array that you would create with the Java code

```
A = new class_name[x1]...[xn];
```

Examples

The following example constructs and populates a 4-by-5 array of `java.lang.Double` objects.

```
dblArray = javaArray ('java.lang.Double', 4, 5);

for m = 1:4
   for n = 1:5
   dblArray(m,n) = java.lang.Double((m*10) + n);
   end
end

dblArray

dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

See Also

`javaObject`, `javaMethod`, `class`, `methodsview`, `isjava`

Purpose Generate an error message based on Java feature support

Syntax

```
javachk(feature)
javachk(feature, component)
```

Description `javachk(feature)` returns a generic error message if the specified Java `feature` is not available in the current MATLAB session. If it is available, `javachk` returns an empty matrix. Possible `feature` arguments are shown in the following table.

Feature	Description
`'awt'`	Abstract Window Toolkit components[1] are available.
`'desktop'`	The MATLAB interactive desktop is running.
`'jvm'`	The Java Virtual Machine is running.
`'swing'`	Swing components[2] are available.

1. Java's GUI components in the Abstract Window Tookit

2. Java's lightweight GUI components in the Java Foundation Classes

`javachk(feature, component)` works the same as the above syntax, except that the specified `component` is also named in the error message. (See the example below.)

Examples The following M-file displays an error with the message `"CreateFrame is not supported on this platform."` when run in a MATLAB session in which the AWT's GUI components are not available. The second argument to `javachk` specifies the name of the M-file, which is then included in the error message generated by MATLAB.

```
javamsg = javachk('awt', mfilename);
if isempty(javamsg)
   myFrame = java.awt.Frame;
   myFrame.setVisible(1);
else
   error(javamsg);
end
```

See Also usejava

javaMethod

Purpose Invokes a Java method

Syntax

```
X = javaMethod('method_name','class_name ,x1,...,xn)
X = javaMethod('method_name',J,x1,...,xn)
```

Description

`javaMethod('method_name','class_name ,x1,...,xn)` invokes the `static` method `method_name` in the class `class_name`, with the argument list that matches `x1,...,xn`.

`javaMethod('method_name',J,x1,...,xn)` invokes the nonstatic method `method_name` on the object J, with the argument list that matches `x1,...,xn`.

Remarks

Using the `javaMethod` function enables you to

- Use methods having names longer than 31 characters
- Specify the method you want to invoke at run-time, for example, as input from an application user

The `javaMethod` function enables you to use methods having names longer than 31 characters. This is the only way you can invoke such a method in MATLAB. For example:

```
javaMethod('DataDefinitionAndDataManipulationTransactions', T);
```

With `javaMethod`, you can also specify the method to be invoked at run-time. In this situation, your code calls `javaMethod` with a string variable in place of the `method name` argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

Note Typically, you do not need to use `javaMethod`. The default MATLAB syntax for invoking a Java method is somewhat simpler and is preferable for most applications. Use `javaMethod` primarily for the two cases described above.

Examples

To invoke the `static` Java method `isNaN` on class, `java.lang.Double`, use

```
javaMethod('isNaN','java.lang.Double',2.2)
```

The following example invokes the nonstatic method `setTitle`, where `frameObj` is a `java.awt.Frame` object.

```
frameObj = java.awt.Frame;
javaMethod('setTitle', frameObj, 'New Title');
```

See Also

`javaArray`, `javaObject`, `import`, `methods`, `isjava`

javaObject

Purpose Constructs a Java object

Syntax

```
J = javaObject('class_name ,x1,...,xn)
```

Description

`javaObject('class_name ,x1,...,xn)` invokes the Java constructor for class `'class_name` with the argument list that matches `x1,...,xn`, to return a new object.

If there is no constructor that matches the class name and argument list passed to `javaObject`, an error occurs.

Remarks

Using the `javaObject` function enables you to

- Use classes having names with more than 31 consecutive characters
- Specify the class for an object at run-time, for example, as input from an application user

The default MATLAB constructor syntax requires that no segment of the input class name be longer than 31 characters. (A *name segment*, is any portion of the class name before, between, or after a period. For example, there are three segments in class, `java.lang.String`.) Any class name segment that exceeds 31 characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use `javaObject` to instantiate the class.

The `javaObject` function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call `javaObject` with a string variable in place of the class name argument.

```
class  = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, you would use

```
strObj = java.lang.String('hello');
```

Note Typically, you will not need to use `javaObject`. The default MATLAB syntax for instantiating a Java class is somewhat simpler and is preferable for most applications. Use `javaObject` primarily for the two cases described above.

Examples

The following example constructs and returns a Java object of class `java.lang.String`:

```
strObj = javaObject('java.lang.String','hello')
```

See Also

`javaArray`, `javaMethod`, `import`, `methods`, `fieldnames`, `isjava`

methods

Purpose

Display method names

Syntax

```
m = methods('classname')
m = methods('object')
m = methods(..., '-full')
```

Description

`m = methods('classname')` returns, in a cell array of strings, the names of all methods for the MATLAB, COM, or Java class, `classname`.

`m = methods('object')` returns the names of all methods for the MATLAB, COM, or Java class of which `object` is an instance.

`m = methods(..., '-full')` returns the full description of the methods defined for the class, including inheritance information and, for COM and Java methods, attributes and signatures. For any overloaded method, the returned array includes a description of each of its signatures.

For MATLAB classes, inheritance information is returned only if that class has been instantiated.

Examples

List the methods of MATLAB class, `stock`:

```
m = methods('stock')
m =
    'display'
    'get'
    'set'
    'stock'
    'subsasgn'
    'subsref'
```

Create a MathWorks sample COM control and list its methods:

```
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
methods(h)

Methods for class com.mwsamp.mwsampctrl.1:

AboutBox          GetR8Array        SetR8             move
Beep              GetR8Vector       SetR8Array        propedit
FireClickEvent    GetVariantArray   SetR8Vector       release
```

```
GetBSTR          GetVariantVector  addproperty      save
GetBSTRArray     Redraw            delete           send
GetI4            SetBSTR           deleteproperty   set
GetI4Array       SetBSTRArray      events
GetI4Vector      SetI4             get
GetIDispatch     SetI4Array        invoke
GetR8            SetI4Vector       load
```

Display a full description of all methods on Java object, `java.awt.Dimension`:

```
methods java.awt.Dimension -full

Dimension(java.awt.Dimension)
Dimension(int,int)
Dimension()
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.lang.Class getClass()  % Inherited from java.lang.Object
            .
            .
```

See Also

`methodsview`, `invoke`, `ismethod`, `help`, `what`, `which`

methodsview

Purpose Displays information on all methods implemented by a class.

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description `methodsview packagename.classname` displays information describing the Java class, `classname`, that is available from the package of Java classes, `packagename`.

`methodsview classname` displays information describing the MATLAB, COM, or imported Java class, `classname`.

`methodsview(object)` displays information describing the `object` instantiated from a COM or Java class.

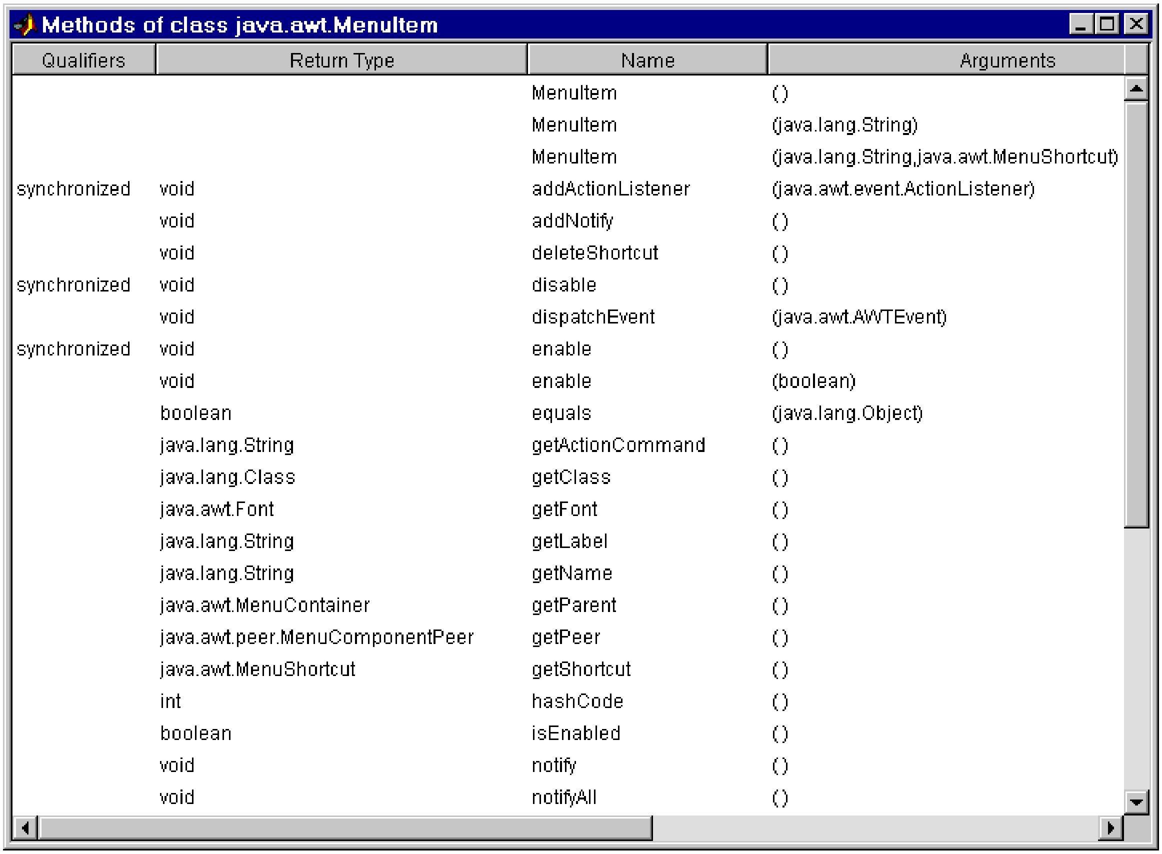
MATLAB creates a new window in response to the `methodsview` command. This window displays all of the methods defined in the specified class. For each of these methods, the following additional information is supplied:

- Name of the method
- Method type qualifiers (for example, `abstract` or `synchronized`)
- Data type returned by the method
- Arguments passed to the method
- Possible exceptions thrown
- Parent of the specified class

Examples The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

MATLAB displays this information in a new window, as shown below



Methods of class java.awt.MenuItem

Qualifiers	Return Type	Name	Arguments
		MenuItem	()
		MenuItem	(java.lang.String)
		MenuItem	(java.lang.String,java.awt.MenuShortcut)
synchronized	void	addActionListener	(java.awt.event.ActionListener)
	void	addNotify	()
	void	deleteShortcut	()
synchronized	void	disable	()
	void	dispatchEvent	(java.awt.AWTEvent)
synchronized	void	enable	()
	void	enable	(boolean)
	boolean	equals	(java.lang.Object)
	java.lang.String	getActionCommand	()
	java.lang.Class	getClass	()
	java.awt.Font	getFont	()
	java.lang.String	getLabel	()
	java.lang.String	getName	()
	java.awt.MenuContainer	getParent	()
	java.awt.peer.MenuComponentPeer	getPeer	()
	java.awt.MenuShortcut	getShortcut	()
	int	hashCode	()
	boolean	isEnabled	()
	void	notify	()
	void	notifyAll	()

See Also

`methods`, `import`, `class`, javaArray

Purpose Determine if a Java feature is supported in MATLAB

Syntax

```
usejava(feature)
```

Description `usejava(feature)` returns 1 if the specified feature is supported and 0 otherwise. Possible `feature` arguments are shown in the following table.

Feature	Description
`'awt'`	Abstract Window Toolkit components[1] are available
`'desktop'`	The MATLAB interactive desktop is running
`'jvm'`	The Java Virtual Machine is running
`'swing'`	Swing components[2] are available

1. Java's GUI components in the Abstract Window Tookit

2. Java's lightweight GUI components in the Java Foundation Classes

Examples The following conditional code ensures that the AWT's GUI components are available before the M-file attempts to display a Java Frame.

```
if usejava('awt')
   myFrame = java.awt.Frame;
else
   disp('Unable to open a Java Frame');
end
```

The next example is part of an M-file that includes Java code. It fails gracefully when run in a MATLAB session that does not have access to a JVM.

```
if ~usejava('jvm')
   error([mfilename ' requires Java to run.']);
end
```

See Also `javachk`

10

COM Functions

`actxcontrol`	Create a COM control in a figure window
`actxserver`	Create a COM automation server
`addproperty (COM)`	Add custom property to COM object
`delete (COM)`	Delete a COM control or server
`deleteproperty (COM)`	Remove custom property from COM object
`eventlisteners (COM)`	Return a list of events attached to listeners
`events (COM)`	Return a list of events that the control can trigger
`fieldnames`	Return property names of a COM object
`get (COM)`	Get property value from an object or interface
`inspect`	Display graphical interface to list and modify property values
`invoke (COM)`	Invoke a method on an object or interface, or display methods
`isevent (COM)`	Determine if an item is an event of a COM control
`ismethod (COM)`	Determine if an item is a method of a COM object

`isprop (COM)`	Determine if an item is a property of a COM object
`load (COM)`	Initialize a COM control object from a file
`methods`	List all methods for the control or server
`methodsview`	Display graphical interface to list method information
`move (COM)`	Resize a COM control in the parent window
`propedit (COM)`	Request the control to display its built-in property page
`registerevent (COM)`	Register an event handler with a control's event
`release (COM)`	Release an interface
`save (COM)`	Serialize a COM control object to a file
`send (COM)`	Obsolete — duplicate of `events`
`set (COM)`	Set an object or interface property to a specific value
`unregisterallevents (COM)`	Unregister all events for a control
`unregisterevent (COM)`	Unregister an event handler with a control's event

Purpose Create a COM control in a figure window

Syntax

```
h = actxcontrol (progid [, position [, fig_handle ...
  [, callback | {event1 eventhandler1; event2 eventhandler2; ...}
  [, filename]]]])
```

Arguments

`progid`
String that is the name of the control to create. The control vendor provides this string.

`position`
Position vector containing the `x` and `y` location and the `xsize` and `ysize` of the control, expressed in pixel units as `[x y xsize ysize]`. Defaults to `[20 20 60 60]`.

`fig_handle`
Handle Graphics handle of the figure window in which the control is to be created. If the control should be invisible, use the handle of an invisible figure window. Defaults to `gcf`.

`callback`
Name of an M-function that accepts a variable number of arguments. This function will be called whenever the control triggers an event. Each argument is converted to a MATLAB string. See the section, "Writing Event Handlers" in the External Interfaces documentation for more information on handling control events.

`event`
Triggered event specified by either number or name.

`eventhandler`
Name of an M-function that accepts a variable number of arguments. This function will be called whenever the control triggers the event associated with it. See "Writing Event Handlers" in the External Interfaces documentation for more information on handling control events.

`filename`
The name of a file to which a previously created control has been saved. When you specify `filename`, MATLAB creates a new control using the `position`, `handle`, and `event`/`eventhandler` arguments, and then initializes the control from the specified file. The `progid` argument in `actxcontrol` must match the `progid` of the saved control.

Description

Create a COM control at a particular location within a figure window. If the parent figure window is invisible, the control will be invisible. The returned COM object represents the default interface for the control. This interface must be released through a call to `release` when it is no longer needed to free the memory and resources used by the interface. Note that releasing the interface does not delete the control itself (use the `delete` command to delete the control.)

The strings specified in the `callback`, `event`, and `eventhandler` arguments are not case sensitive.

Note There are two ways to handle events. You can create a single handler (`callback`) for all events, or you can specify a cell array that contains pairs of events and event handlers. In the cell array format, specify events by name in a quoted string. There is no limit to the number of pairs that can be specified in the cell array. Although using the single callback method may be easier in some cases, using the cell array technique creates more efficient code that results in better performance.

For an example callback event handler, see the file `sampev.m` in the `toolbox\matlab\winfun\comcli` directory.

Examples

Basic Control Methods

Create a control that runs Microsoft's Calendar application:

```
f = figure('pos',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f)
cal =
    COM.mscal.calendar
```

Call the `get` method on `cal` to list all properties of the Calendar:

```
get(cal)
             BackColor: 2.1475e+009
                   Day: 23
               DayFont: [1x1 Interface.mscal.calendar.DayFont]
                 Value: '8/20/2001'
                     .
                     .
```

Read just one property to record today's date:

```
date = get(cal, 'Value')
date =
   8/23/2001
```

Set the Day property to a new value:

```
set(cal, 'Day', 5);
date = get(cal, 'Value')
date =
   8/5/2001
```

Calling invoke with no arguments lists all available methods:

```
meth = invoke(cal)
meth =
          NextDay: 'HRESULT NextDay(handle)'
        NextMonth: 'HRESULT NextMonth(handle)'
         NextWeek: 'HRESULT NextWeek(handle)'
         NextYear: 'HRESULT NextYear(handle)'
            .
            .
```

Invoke the NextWeek method to advance the current date by one week:

```
NextWeek(cal);
date = get(cal, 'Value')
date =
   8/12/2001
```

Call events to list all Calendar events that can be triggered:

```
events(cal)
ans =
   Click = void Click()
   DblClick = void DblClick()
   KeyDown = void KeyDown(int16 KeyCode, int16 Shift)
   KeyPress = void KeyPress(int16 KeyAscii)
   KeyUp = void KeyUp(int16 KeyCode, int16 Shift)
   BeforeUpdate = void BeforeUpdate(int16 Cancel)
   AfterUpdate = void AfterUpdate()
   NewMonth = void NewMonth()
```

```
NewYear = void NewYear()
```

Set Up Event Handling

See the section, Sample Event Handlers in the External Interfaces documentation for examples of event handler functions and how to register them with MATLAB.

See Also

`actxserver`, `release`, `delete`, `save`, `load`

Purpose

Create a COM Automation server and return a COM object for the server's default interface

Syntax

```
h = actxserver (progid [, machinename])
```

Arguments

`progid`
This is a string that is the name of the control to instantiate. This string is provided by the control or server vendor and should be obtained from the vendor's documentation. For example, the `progid` for MATLAB is `matlab.application`.

`machinename`
This is the name of a remote machine on which the server is to be run. This argument is optional and is used only in environments that support Distributed Component Object Model (DCOM) — see "Using MATLAB As a DCOM Server Client" in the External Interfaces documentation. This can be an IP address or a DNS name.

Description

Create a COM Automation server and return a COM object that represents the server's default interface. Local/Remote servers differ from controls in that they are run in a separate address space (and possibly on a separate machine) and are not part of the MATLAB process. Additionally, any user interface that they display will be in a separate window and will not be attached to the MATLAB process. Examples of local servers are Microsoft Excel and Microsoft Word. There is currently no support for events generated from automation servers.

Examples

Launch Microsoft Excel and make the main frame window visible:

```
e = actxserver ('Excel.Application')
e =
    COM.excel.application
set(e, 'Visible', 1);
```

Call the get method on the excel object to list all properties of the application:

```
get(e)
ans =
      Application: [1x1 Interface.excel.application.Application]
           Creator: 'xlCreatorCode'
            Parent: [1x1 Interface.Excel.Application.Parent]
         Workbooks: [1x1 Interface.excel.application.Workbooks]
      UsableHeight: 666.7500
                .
                .
```

Create an interface:

```
eWorkbooks = get(e, 'Workbooks')
eWorkbooks =
    Interface.excel.application.Workbooks
```

List all methods for that interface by calling invoke with just the handle argument:

```
invoke(eWorkbooks)
ans =
         Add: 'handle Add(handle, [Optional]Variant)'
       Close: 'void Close(handle)'
        Item: 'handle Item(handle, Variant)'
        Open: 'handle Open(handle, string, [Optional]Variant)'
    OpenText: 'void OpenText(handle, string, [Optional]Variant)'
```

Invoke the Add method on workbooks to add a new workbook, also creating a new interface:

```
w = Add(eWorkbooks)
w =
    Interface.Excel.Application.Workbooks.Add
```

Quit the application and delete the object:

```
Quit(e);
delete(e);
```

See Also actxcontrol, release, delete, save, load

Purpose Add custom property to COM object

Syntax

```
addproperty(h, 'propertyname')
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`
A string specifying the name of the custom property to add to the object or interface.

Description Add a custom property, `propertyname`, to the object or interface, `h`. You can assign a value to that property using `set`.

Examples Create an `mwsamp` control and add a new property named `Position` to it. Assign an array value to the property:

```
f = figure('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
get(h)
     Label: 'Label'
    Radius: 20

addproperty(h, 'Position');
set(h, 'Position', [200 120]);
get(h)
       Label: 'Label'
      Radius: 20
    Position: [200 120]

get(h, 'Position')
ans =
   200   120
```

See Also `deleteproperty`, `get`, `set`, `inspect`

delete (COM)

Purpose Delete a COM control or server

Syntax

```
delete(h)
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

Description Release all interfaces derived from the specified COM server or control, and then delete the server or control itself. This is different from releasing an interface, which releases and invalidates only that interface.

Examples Create a Microsoft Calender application. Then create a `TitleFont` interface and use it to change the appearance of the font of the calendar's title:

```
f = figure('pos',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);

TFont = get(cal, 'TitleFont')
TFont =
   Interface.mscal.calendar.TitleFont

set(TFont, 'Name', 'Viva BoldExtraExtended');
set(TFont, 'Bold', 0);
```

When you're finished working with the title font, release the `TitleFont` interface:

```
release(TFont);
```

Now create a `GridFont` interface and use it to modify the size of the calendar's date numerals:

```
GFont = get(cal, 'GridFont')
GFont =
   Interface.mscal.calendar.GridFont

set(GFont, 'Size', 16);
```

When you're done, delete the `cal` object and the figure window. Deleting the `cal` object also releases all interfaces to the object (e.g., `GFont`):

```
delete(cal);
delete(f);
clear f;
```

Note that, although the object and interfaces themselves have been destroyed, the variables assigned to them still reside in the MATLAB workspace until you remove them with `clear`.

```
whos
  Name        Size                     Bytes  Class

  GFont       1x1                          0  handle
  TFone       1x1                          0  handle
  cal         1x1                          0  handle

Grand total is 3 elements using 0 bytes
```

See Also `release`, `save`, `load`, `actxcontrol`, `actxserver`

deleteproperty (COM)

Purpose Remove custom property from COM object

Syntax `deleteproperty(h, 'propertyname')`

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`
A string specifying the name of the custom property to delete.

Description Delete a property, `propertyname`, from the custom properties belonging to object or interface, `h`. You can only delete properties that have been created with `addproperty`.

Examples Create an `mwsamp` control and add a new property named `Position` to it. Assign an array value to the property:

```
f = figure('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
get(h)
     Label: 'Label'
    Radius: 20

addproperty(h, 'Position');
set(h, 'Position', [200 120]);
get(h)
       Label: 'Label'
      Radius: 20
    Position: [200 120]
```

Delete the custom `Position` property:

```
deleteproperty(h, 'Position');
get(h)
     Label: 'Label'
    Radius: 20
```

See Also `addproperty`, `get`, `set`, `inspect`

Purpose Return a list of events attached to listeners

Syntax

```
eventlisteners(h)
```

Arguments

`h`
Handle for a MATLAB COM control object.

Description

`eventlisteners` lists any events, along with their callback or event handler routines, that have been registered with control, `h`. The function returns a cell array of strings, with each row containing the name of a registered event and the handler routine for that event. If the control has no registered events, then `eventlisteners` returns an empty cell array.

Events and their callback or event handler routines must be registered in order for the control to respond to them. You can register events either when you create the control, using `actxcontrol`, or at any time afterwards, using `registerevent`.

Examples

Create an `mwsamp` control, registering only the `Click` event. `eventlisteners` returns the name of the event and its event handler routine, `myclick`:

```
f = figure('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'});

eventlisteners(h)
ans =
    'click'    'myclick'
```

Register two more events: `DblClick` and `MouseDown`. `eventlisteners` returns the names of the three registered events along with their respective handler routines:

```
registerevent(h, {'DblClick', 'my2click'; ...
    'MouseDown' 'mymoused'});

eventlisteners(h)
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
    'mousedown'    'mymoused'
```

Now unregister all events for the control, and `eventlisteners` returns an empty cell array, indicating that no events have been registered for the control:

```
unregisterallevents(h)

eventlisteners(h)
ans =
     {}
```

See Also

`events`, `registerevent`, `unregisterevent`, `unregisterallevents`, `isevent`

Purpose Return a list of events that the control can trigger

Syntax

```
events(h)
```

Arguments

`h`
Handle for a MATLAB COM control object.

Description Returns a structure array containing all events, both registered and unregistered, known to the control, and the function prototype used when calling the event handler routine. For each array element, the structure field is the event name and the contents of that field is the function prototype for that event's handler.

Note The `send` function is identical to `events`, but `send` will be made obsolete in a future release.

Examples Create an `mwsamp` control and list all events:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

events(h)
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

Or assign the output to a variable and get one field of the returned structure:

```
ev = events(h);

ev.MouseDown
ans =
void MouseDown(int16 Button, int16 Shift, Variant x, Variant y)
```

See Also `isevent`, `eventlisteners`, `registerevent`, `unregisterevent`, `unregisterallevents`

fieldnames

Purpose Return field names of a structure, or property names of an object

Syntax

```
names = fieldnames(s)
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Description

`names = fieldnames(s)` returns a cell array of strings containing the structure field names associated with the structure `s`.

`names = fieldnames(obj)` returns a cell array of strings containing the names of the public data fields associated with `obj`, which is either a MATLAB, COM, or Java object.

`names = fieldnames(obj,'-full')` returns a cell array of strings containing the name, type, attributes, and inheritance of each field associated with `obj`, which is either a MATLAB, COM, or Java object.

Examples

Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1
```

the command `n = fieldnames(mystr)` yields

```
n =
    'name'
    'ID'
```

In another example, if `f` is an object of Java class `java.awt.Frame`, the command `fieldnames(f)` lists the properties of `f`.

```
f = java.awt.Frame;

fieldnames(f)
ans =
    'WIDTH'
    'HEIGHT'
    'PROPERTIES'
    'SOMEBITS'
    'FRAMEBITS'
```

```
'ALLBITS'
    .
    .
```

See Also

`isfield`, `orderfields`, `rmfield`, dynamic field names

get (COM)

Purpose Retrieve a property value from an interface or get a list of properties

Syntax

```
v = get(h[, 'propertyname'])
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`
A string that is the name of the property value to be retrieved.

Description Returns the value of the property specified by `propertyname`. If no property is specified, then `get` returns a list of all properties for the object or interface.

The meaning and type of the return value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. See "Converting Data" in the External Interfaces documentation for a description of how MATLAB converts COM data types.

Examples Create a COM server running Microsoft Excel:

```
e = actxserver ('Excel.Application');
```

Retrieve a single property value:

```
get(e, 'Path')
ans =
   D:\Applications\MSOffice\Office
```

Retrieve a list of all properties for the `CommandBars` interface:

```
c = get(e, 'CommandBars');
get(c)
ans =
              Application: [1x1
Interface.excel.application.CommandBars.Application]
                  Creator: 1.4808e+009
            ActionControl: []
            ActiveMenuBar: [1x1
Interface.excel.application.CommandBars.ActiveMenuBar]
                    Count: 94
```

```
            DisplayTooltips: 1
      DisplayKeysInTooltips: 0
               LargeButtons: 0
         MenuAnimationStyle: 'msoMenuAnimationNone'
                     Parent: [1x1
Interface.excel.application.CommandBars.Parent]
              AdaptiveMenus: 0
               DisplayFonts: 1
```

See Also

`set`, `inspect`, `isprop`, `addproperty`, `deleteproperty`

inspect

Purpose

Display graphical user interface to list and modify property values

Syntax

```
inspect
inspect(h)
```

Description

`inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor.

`inspect(h)` creates a Property Inspector window for the graphics, Java, or COM object attached to handle, `h`.

To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

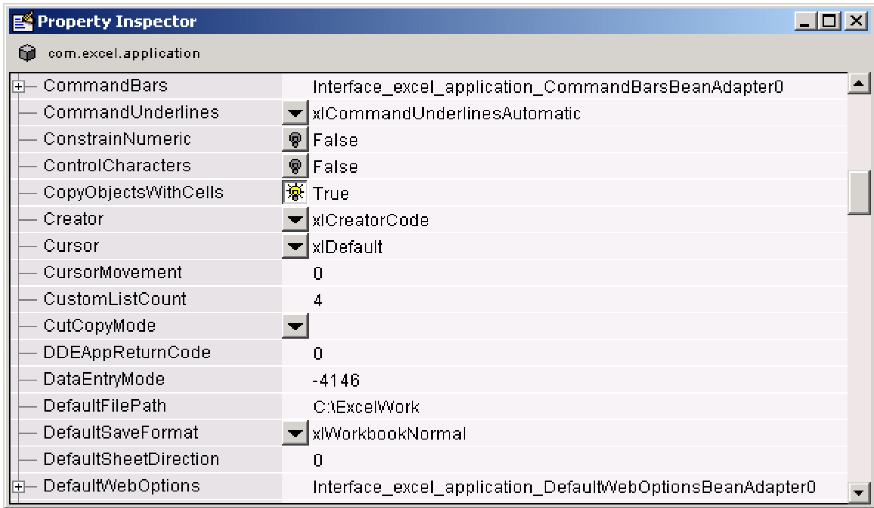
Note If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

Example

Create a COM Excel server and open a Property Inspector window with `inspect`:

```
h = actxserver('excel.application');
inspect(h)
```

Scroll down until you see the `DefaultFilePath` property. Click on the property name shown at the left. Then replace the text at the right with `C:\ExcelWork`.



Check this field in the MATLAB command window and confirm that it has changed:

```
get(h, 'DefaultFilePath')
ans =
   C:\ExcelWork
```

See Also

`get`, `set`, `isprop`, `guide`, `addproperty`, `deleteproperty`

invoke (COM)

Purpose Invoke a method on an object or interface

Syntax

```
v = invoke(h, ['methodname' [, arg1, arg2, ...]])
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`methodname`
A string that is the name of the method to be invoked.

`arg1, ..., argn`
Arguments, if any, required by the method being invoked.

Description Invoke a method on an object's interface and retrieve the return value of the method, if any. The data type of the value is dependent upon the specific method being invoked and is determined by the specific control or server. If the method returns a COM interface, then `invoke` returns a new MATLAB COM object that represents the interface returned. See "Converting Data" in the External Interfaces documentation for a description of how MATLAB converts COM data types.

When you specify only a handle argument with invoke, MATLAB returns a structure array containing a list of all methods available for the object and their prototypes.

Examples Create an `mwsamp` control and invoke its `Redraw` method:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

set(h, 'Radius', 100);
invoke(h, 'Redraw');
```

Here is a simpler way to `invoke`. Just call the method directly, passing the handle, and any arguments:

```
Redraw(h);
```

Call `invoke` with only the handle argument to display a list of all `mwsamp` methods:

```
invoke(h)
```

```
ans =
                Beep: 'void Beep(handle)'
              Redraw: 'void Redraw(handle)'
     GetVariantArray: 'Variant GetVariantArray(handle)'
             .
             .
           etc.
```

See Also

methods, ismethod

isevent (COM)

Purpose

Determine if an item is an event of a COM control

Syntax

```
isevent(h, 'name')
```

Arguments

`h`
Handle for a MATLAB COM control object.

`name`
Name of the item to test.

Description

Returns a logical `1` (`true`) if the specified `name` is an event that can be recognized and responded to by the control, `h`. Otherwise, `isevent` returns logical `0` (`false`).

`isevent` returns the same value regardless of whether the specified event is registered with the control or not. In order for the control to respond to the event, you must first register the event using either `actxcontrol` or `registerevent`.

The string specified in the `name` argument is not case sensitive.

Examples

Create an `mwsamp` control and test to see if `DblClick` is an event recognized by the control. `isevent` returns `true`:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

isevent(h, 'DblClick')
ans =
     1
```

Try the same test on `Redraw`, which is a method, and `isevent` returns `false`:

```
isevent(h, 'Redraw')
ans =
     0
```

See Also

`events`, `eventlisteners`, `registerevent`, `unregisterevent`, `unregisterallevents`

Purpose

Determine if an item is a method of a COM object

Syntax

```
ismethod(h, 'name')
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`name`
Name of the item to test.

Description

Returns a logical `1` (`true`) if the specified `name` is a method that you can call on COM object, `h`. Otherwise, `ismethod` returns logical `0` (`false`).

Examples

Create an Excel application and test to see if `SaveWorkspace` is a method of the object. `ismethod` returns `true`:

```
h = actxserver ('Excel.Application');

ismethod(h, 'SaveWorkspace')
ans =
     1
```

Try the same test on `UsableWidth`, which is a property, and `isevent` returns `false`:

```
ismethod(h, 'UsableWidth')
ans =
     0
```

See Also

`methods`, `invoke`

isprop (COM)

Purpose Determine if an item is a property of a COM object

Syntax

```
isprop(h, 'name')
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`name`
Name of the item to test.

Description Returns a logical `1` (`true`) if the specified `name` is a property you can use with COM object, `h`. Otherwise, `isprop` returns logical `0` (`false`).

Examples Create an Excel application and test to see if `UsableWidth` is a property of the object. `isprop` returns `true`:

```
h = actxserver ('Excel.Application');

isprop(h, 'UsableWidth')
ans =
     1
```

Try the same test on `SaveWorkspace`, which is a method, and `isprop` returns `false`:

```
isprop(h, 'SaveWorkspace')
ans =
     0
```

See Also `get`, `inspect`, `addproperty`, `deleteproperty`

load (COM)

Purpose Initialize a COM object from a file

Syntax

```
load(h, 'filename')
```

Arguments

`h`
Handle for a MATLAB COM control object.

`filename`
The full path and filename of the serialized data.

Description Initializes the COM object associated with the interface represented by the MATLAB COM object `h` from a file. The file must have been created previously by serializing an instance of the same control.

The COM `load` function is only supported for controls at this time.

Examples Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
save(h, 'mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
set(h, 'Label', 'Circle');
set(h, 'Radius', 50);
Redraw(h);
```

Using the `load` function, you can restore the control to its original state:

```
load(h, 'mwsample');
get(h)
ans =
     Label: 'Label'
    Radius: 20
```

See Also `save`, `actxcontrol`, `actxserver`, `release`, `delete`

methods

Purpose

Display method names

Syntax

```
m = methods('classname')
m = methods('object')
m = methods(..., '-full')
```

Description

`m = methods('classname')` returns, in a cell array of strings, the names of all methods for the MATLAB, COM, or Java class, `classname`.

`m = methods('object')` returns the names of all methods for the MATLAB, COM, or Java class of which `object` is an instance.

`m = methods(..., '-full')` returns the full description of the methods defined for the class, including inheritance information and, for COM and Java methods, attributes and signatures. For any overloaded method, the returned array includes a description of each of its signatures.

For MATLAB classes, inheritance information is returned only if that class has been instantiated.

Examples

List the methods of MATLAB class, `stock`:

```
m = methods('stock')
m =
    'display'
    'get'
    'set'
    'stock'
    'subsasgn'
    'subsref'
```

Create a MathWorks sample COM control and list its methods:

```
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
methods(h)

Methods for class com.mwsamp.mwsampctrl.1:

AboutBox          GetR8Array        SetR8             move
Beep              GetR8Vector       SetR8Array        propedit
FireClickEvent    GetVariantArray   SetR8Vector       release
```

```
GetBSTR            GetVariantVector  addproperty       save
GetBSTRArray       Redraw            delete            send
GetI4              SetBSTR           deleteproperty    set
GetI4Array         SetBSTRArray      events
GetI4Vector        SetI4             get
GetIDispatch       SetI4Array        invoke
GetR8              SetI4Vector       load
```

Display a full description of all methods on Java object, `java.awt.Dimension`:

```
methods java.awt.Dimension -full

Dimension(java.awt.Dimension)
Dimension(int,int)
Dimension()
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.lang.Class getClass()  % Inherited from java.lang.Object
            .
            .
```

See Also `methodsview`, `invoke`, `ismethod`, `help`, `what`, `which`

methodsview

Purpose

Displays information on all methods implemented by a class.

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description

`methodsview packagename.classname` displays information describing the Java class, `classname`, that is available from the package of Java classes, `packagename`.

`methodsview classname` displays information describing the MATLAB, COM, or imported Java class, `classname`.

`methodsview(object)` displays information describing the `object` instantiated from a COM or Java class.

MATLAB creates a new window in response to the `methodsview` command. This window displays all of the methods defined in the specified class. For each of these methods, the following additional information is supplied:

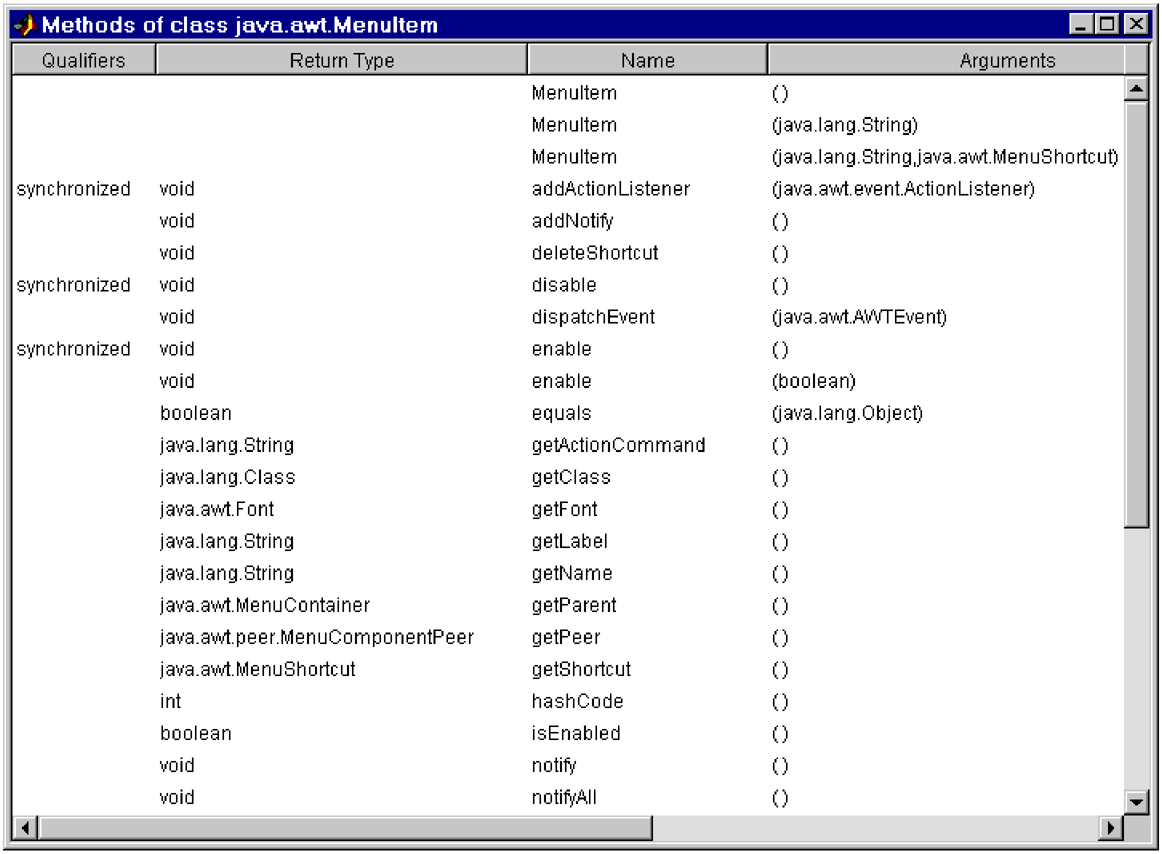
- Name of the method
- Method type qualifiers (for example, `abstract` or `synchronized`)
- Data type returned by the method
- Arguments passed to the method
- Possible exceptions thrown
- Parent of the specified class

Examples

The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

MATLAB displays this information in a new window, as shown below



Methods of class java.awt.MenuItem

Qualifiers	Return Type	Name	Arguments
		MenuItem	()
		MenuItem	(java.lang.String)
		MenuItem	(java.lang.String,java.awt.MenuShortcut)
synchronized	void	addActionListener	(java.awt.event.ActionListener)
	void	addNotify	()
	void	deleteShortcut	()
synchronized	void	disable	()
	void	dispatchEvent	(java.awt.AWTEvent)
synchronized	void	enable	()
	void	enable	(boolean)
	boolean	equals	(java.lang.Object)
	java.lang.String	getActionCommand	()
	java.lang.Class	getClass	()
	java.awt.Font	getFont	()
	java.lang.String	getLabel	()
	java.lang.String	getName	()
	java.awt.MenuContainer	getParent	()
	java.awt.peer.MenuComponentPeer	getPeer	()
	java.awt.MenuShortcut	getShortcut	()
	int	hashCode	()
	boolean	isEnabled	()
	void	notify	()
	void	notifyAll	()

See Also

`methods`, `import`, `class`, javaArray

move (COM)

Purpose

Move and/or resize a COM control in its parent window

Syntax

```
move(h, position)
```

Arguments

`h`
Handle for a MATLAB COM control object.

`position`
A four-element vector specifying the position of the control in the parent window. The elements of the vector are

```
[left, bottom, width, height]
```

Description

Moves the control to the position specified by the `position` argument. When you use move with only the handle argument, h, it returns a four-element vector indicating the current position of the control.

Examples

This example moves the control:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
pos = move(h, [50 50 200 200])
pos =
    50    50   200   200
```

The next example resizes the control to always be centered in the figure as you resize the figure window. Start by creating the script `resizectrl.m` that contains

```
% Get the new position and size of the figure window
  fpos = get(gcbo, 'position');

% Resize the control accordingly
  move(h, [0 0 fpos(3) fpos(4)]);
```

Now execute the following in MATLAB or in an M-file:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
set(f, 'ResizeFcn', 'resizectrl');
```

As you resize the figure window, notice that the circle moves so that it is always positioned in the center of the window.

See Also set, get

propedit (COM)

Purpose

Request the control to display its built-in property page

Syntax

`propedit(h)`

Arguments

`h`
Handle for a MATLAB COM control object.

Description

Request the control to display its built-in property page. Note that some controls do not have a built-in property page. For those objects, this command will fail.

Examples

Create a Microsoft Calendar control and display its property page:

```
cal = actxcontrol('mscal.calendar', [0 0 500 500]);
propedit(cal)
```

See Also

`inspect`, `get`

Purpose

Register an event handler with a control's event

Syntax

```
registerevent(h, callback |
   {event1 eventhandler1; event2 eventhandler2; ...})
```

Arguments

`h`
Handle for a MATLAB COM control object.

`callback`
Name of an M-function that accepts a variable number of arguments. This function will be called whenever the control triggers an event. Each argument is converted to a MATLAB string. See the section, "Writing Event Handlers" in the External Interfaces/API documentation for more information on handling control events.

`event`
Any event associated with `h` that can be triggered. Specify `event` using the event name.

`eventhandler`
Name of an M-function that accepts a variable number of arguments. This function will be called whenever the control triggers the event associated with it. See "Writing Event Handlers" in the External Interfaces/API documentation for more information on handling control events.

Description

Register one or more events with a single callback function or with a separate handler function for each event. You can either register events at the time you create the control (using `actxcontrol`), or register them dynamically at any time after the control has been created (using `registerevent`).

The strings specified in the `callback`, `event`, and `eventhandler` arguments are not case sensitive.

Note There are two ways to handle events. You can create a single handler (`callback`) for all events, or you can specify a cell array that contains pairs of events and event handlers. In the cell array format, specify events by name in a quoted string. There is no limit to the number of pairs that can be specified in the cell array. Although using the single callback method may be easier in some cases, using the cell array technique creates more efficient code that results in better performance.

Examples

Create an `mwsamp` control and list all events associated with the control:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

events(h)
ans =
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

Register all events with the same callback routine, `sampev`. Use the `eventlisteners` function to see the event handler used by each event:

```
registerevent(h, 'sampev');
eventlisteners(h)
ans =
    'click'         'sampev'
    'dblclick'      'sampev'
    'mousedown'     'sampev'

unregisterallevents(h);
```

Register the `Click` and `DblClick` events with event handlers `myclick` and `my2click`, respectively:

```
registerevent(h, {'click' 'myclick'; 'dblclick' 'my2click'});
eventlisteners(h)
ans =
    'click'        'myclick'
```

'dblclick'	'my2click'

See Also

events, eventlisteners, unregisterevent, unregisterallevents, isevent

release (COM)

Purpose Release an interface

Syntax

```
release(h)
```

Arguments

`h`
Handle for a COM object that represents the interface to be released.

Description Release the interface and all resources used by the interface. Each interface handle must be released when you are finished manipulating its properties and invoking its methods. Once an interface has been released, it is no longer valid and subsequent operations on the MATLAB object that represents that interface will result in errors.

Note Releasing the interface will not delete the control itself (see `delete`), since other interfaces on that object may still be active. See “Releasing Interfaces” in the External Interfaces/API documentation for more information.

Examples Create a Microsoft Calender application. Then create a `TitleFont` interface and use it to change the appearance of the font of the calendar’s title:

```
f = figure('pos',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);

TFont = get(cal, 'TitleFont')
TFont =
   Interface.mscal.calendar.TitleFont

set(TFont, 'Name', 'Viva BoldExtraExtended');
set(TFont, 'Bold', 0);
```

When you’re finished working with the title font, release the `TitleFont` interface:

```
release(TFont);
```

Now create a `GridFont` interface and use it to modify the size of the calendar’s date numerals:

```
GFont = get(cal, 'GridFont')
```

```
GFont =
   Interface.mscal.calendar.GridFont

set(GFont, 'Size', 16);
```

When you're done, delete the `cal` object and the figure window:

```
delete(cal);
delete(f);
clear f;
```

See Also

`delete`, `save`, `load`, `actxcontrol`, `actxserver`

save (COM)

Purpose Serialize a COM control object to a file

Syntax

```
save(h, 'filename')
```

Arguments

`h`
Handle for a MATLAB COM control object.

`filename`
The full path and filename of the serialized data.

Description Save the COM control object associated with the interface represented by the MATLAB COM object `h` into a file.

The COM `save` function is only supported for controls at this time.

Examples Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
save(h, 'mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
set(h, 'Label', 'Circle');
set(h, 'Radius', 50);
Redraw(h);
```

Using the `load` function, you can restore the control to its original state:

```
load(h, 'mwsample');
get(h)
ans =
      Label: 'Label'
     Radius: 20
```

See Also `load`, `actxcontrol`, `actxserver`, `release`, `delete`

Purpose Return a list of events that the control can trigger

Note Support for `send` will be removed in a future release of MATLAB. Use the `events` function instead of `send`.

---

set (COM)

Purpose Set an interface property to a specific value

Syntax

```
set(h, 'propertyname', value[, 'propertyname2', value2, ...])
```

Arguments

`h`
Handle for a COM object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`
A string that is the name of the property to be set.

`value`
The value to which the interface property is set.

Description Set one or more properties of a COM object to the specified value(s). Each `propertyname` argument must be followed by a `value` argument.

See “Converting Data” in the External Interfaces documentation for information on how MATLAB converts workspace matrices to COM data types.

Examples Create an `mwsamp` control and use `set` to change the `Label` and `Radius` properties:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

set(h, 'Label', 'Click to fire event', 'Radius', 40);
invoke(h, 'Redraw');
```

See Also `get`, `inspect`, `isprop`, `addproperty`, `deleteproperty`

Purpose Unregister all events for a control

Syntax `unregisterallevents(h)`

Arguments `h`
Handle for a MATLAB COM control object.

Description Unregister all events that have previously been registered with control, `h`. After calling `unregisterallevents`, the control will no longer respond to any events until you register them again using the `registerevent` function.

Examples Create an `mwsamp` control, registering three events and their respective handler routines. Use the `eventlisteners` function to see the event handler used by each event:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
    'MouseDown' 'mymoused'});

eventlisteners(h)
ans =
    'click'          'myclick'
    'dblclick'       'my2click'
    'mousedown'      'mymoused'
```

Unregister all of these events at once with `unregisterallevents`. Now, calling `eventlisteners` returns an empty cell array, indicating that there are no longer any events registered with the control:

```
unregisterallevents(h);
eventlisteners(h)
ans =
     {}
```

To unregister specific events, use the unregisterevent function:

```
unregisterevent(h, {'click' 'myclick'; 'dblclick' 'my2click'});
eventlisteners(h)
ans =
     {}
```

See Also

events, eventlisteners, registerevent, unregisterevent, isevent

Purpose

Unregister an event handler with a control's event

Syntax

```
unregisterevent(h, callback |
   {event1 eventhandler1; event2 eventhandler2; ...})
```

Arguments

`h`
Handle for a MATLAB COM control object.

`callback`
Name of an M-function previously registered with this object to handle events. Callbacks are registered using either `actxcontrol` or `registerevent`.

`event`
Any event associated with `h` that can be triggered. Specify `event` using the event name. Unlike `actxcontrol`, `unregisterevent` does not accept numeric event identifiers.

`eventhandler`
Name of the event handler routine that you want to unregister for the event specified in the preceding `event` argument.

Description

Unregister the specified `callback` routines with all events for this control, or unregister each specified `eventhandler` routine with the `event` associated with it in the argument list. Once you unregister a callback or event handler routine, MATLAB no longer responds to the event using that routine.

The strings specified in the `callback`, `event`, and `eventhandler` arguments are not case sensitive.

You can unregister events at any time after a control has been created.

Examples

Create an `mwsamp` control and register all events with the same callback routine, `sampev`. Use the `eventlisteners` function to see the event handler used by each event. In this case, each event, when fired, will call `sampev.m`:

```
f = figure ('pos', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    'sampev');

eventlisteners(h)
ans =
    'click'        'sampev'
```

```
        'dblclick'      'sampev'
        'mousedown'     'sampev'
```

Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, you see that `dblclick` is no longer registered. The control will no longer respond when you double-click the mouse over it:

```
unregisterevent(h, {'dblclick' 'sampev'});
eventlisteners(h)
ans =
    'click'         'sampev'
    'mousedown'     'sampev'
```

This time, register the `click` and `dblclick` events with a different event handler for each: `myclick` and `my2click`, respectively:

```
registerevent(h, {'click' 'myclick'; 'dblclick' 'my2click'});
eventlisteners(h)
ans =

    'click'        'myclick'
    'dblclick'     'my2click'
```

You can unregister these same events by specifying event names and their handler routines in a cell array. Note that `eventlisteners` now returns an empty cell array, meaning that no events are registered for the `mwsamp` control:

```
unregisterevent(h, {'click' 'myclick'; 'dblclick' 'my2click'});
eventlisteners(h)
ans =
     {}
```

In this last example, you could have used `unregisterallevents` instead:

```
unregisterallevents(h);
```

See Also `events`, `eventlisteners`, `registerevent`, `unregisterallevents`, `isevent`

11

DDE Functions

`ddeadv`	Set up advisory link
`ddeexec`	Send string for execution
`ddeinit`	Initiate DDE conversation
`ddepoke`	Send data to application
`ddereq`	Request data from application
`ddeterm`	Terminate DDE conversation
`ddeunadv`	Release advisory link

ddeadv

Purpose

Set up advisory link

Syntax

```
rc = ddeadv(channel,'item','callback')
rc = ddeadv(channel,'item','callback','upmtx')
rc = ddeadv(channel,'item','callback','upmtx',format)
rc = ddeadv(channel,'item','callback','upmtx',format,timeout)
```

Description

`ddeadv` sets up an advisory link between MATLAB and a server application. When the data identified by the `item` argument changes, the string specified by the `callback` argument is passed to the `eval` function and evaluated. If the advisory link is a hot link, DDE modifies `upmtx`, the update matrix, to reflect the data in `item`.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddeadv` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
`callback`	String specifying the callback that is evaluated on update notification. Changing the data identified by `item` at the server causes `callback` to get passed to the `eval` function to be evaluated.
`upmtx` (*optional*)	String specifying the name of a matrix that holds data sent with an update notification. If `upmtx` is included, changing `item` at the server causes `upmtx` to be updated with the revised data. Specifying `upmtx` creates a hot link. Omitting `upmtx` or specifying it as an empty string creates a warm link. If `upmtx` exists in the workspace, its contents are overwritten. If `upmtx` does not exist, it is created.

`format` (*optional*) — Two-element array specifying the format of the data to be sent on update. The first element specifies the Windows clipboard format to use for the data. The only currently supported format is `cf_text`, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `numeric` (the default, which corresponds to a value of `0`) and `string` (which corresponds to a value of `1`). The default format array is `[1 0]`.

`timeout` (*optional*) — Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). If advisory link is not established within `timeout` milliseconds, the function fails. The default value of `timeout` is three seconds.

Examples

Set up a hot link between a range of cells in Excel (Row 1, Column 1 through Row 5, Column 5) and the matrix `x`. If successful, display the matrix:

```
rc = ddeadv(channel, 'r1c1:r5c5', 'disp(x)', 'x');
```

Communication with Excel must have been established previously with a `ddeinit` command.

See Also

`ddeexec`, `ddeinit`, `ddepoke`, `ddereq`, `ddeterm`, `ddeunadv`

ddeexec

Purpose Send string for execution

Syntax

```
rc = ddeexec(channel,'command')
rc = ddeexec(channel,'command','item')
rc = ddeexec(channel,'command','item',timeout)
```

Description `ddeexec` sends a string for execution to another application via an established DDE conversation. Specify the string as the `command` argument.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddeexec` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`command`	String specifying the command to be executed.
`item` (*optional*)	String specifying the DDE item name for execution. This argument is not used for many applications. If your application requires this argument, it provides additional information for `command`. Consult your server documentation for more information.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples Given the channel assigned to a conversation, send a command to Excel:

```
rc = ddeexec(channel,'[formula.goto("r1c1")]')
```

Communication with Excel must have been established previously with a `ddeinit` command.

See Also `ddeadv`, `ddeinit`, `ddepoke`, `ddereq`, `ddeterm`, `ddeunadv`

Purpose Initiate DDE conversation

Syntax

```
channel = ddeinit('service','topic')
```

Description `channel = ddeinit('service','topic')` returns a channel handle assigned to the conversation, which is used with other MATLAB DDE functions. `'service'` is a string specifying the service or application name for the conversation. `'topic'` is a string specifying the topic for the conversation.

Examples To initiate a conversation with Excel for the spreadsheet `'stocks.xls'`:

```
channel = ddeinit('excel','stocks.xls')

channel =
        0.00
```

See Also `ddeadv`, `ddeexec`, `ddepoke`, `ddereq`, `ddeterm`, `ddeunadv`

ddepoke

Purpose

Send data to application

Syntax

```
rc = ddepoke(channel,'item',data)
rc = ddepoke(channel,'item',data,format)
rc = ddepoke(channel,'item',data,format,timeout)
```

Description

`ddepoke` sends data to an application via an established DDE conversation. `ddepoke` formats the data matrix as follows before sending it to the server application:

- String matrices are converted, element by element, to characters and the resulting character buffer is sent.
- Numeric matrices are sent as tab-delimited columns and carriage-return, line-feed delimited rows of numbers. Only the real part of nonsparse matrices are sent.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddepoke` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the DDE item for the data sent. Item is the server data entity that is to contain the data sent in the `data` argument.
`data`	Matrix containing the data to send.
`format` (*optional*)	Scalar specifying the format of the data requested. The value indicates the Windows clipboard format to use for the data transfer. The only format currently supported is `cf_text`, which corresponds to a value of `1`.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Assume that a conversation channel with Excel has previously been established with `ddeinit`. To send a 5-by-5 identity matrix to Excel, placing the data in Row 1, Column 1 through Row 5, Column 5:

```
rc = ddepoke(channel, 'r1c1:r5c5', eye(5));
```

See Also

`ddeadv`, `ddeexec`, `ddeinit`, `ddereq`, `ddeterm`, `ddeunadv`

ddereq

Purpose Request data from application

Syntax

```
data = ddereq(channel,'item')
data = ddereq(channel,'item',format)
data = ddereq(channel,'item',format,timeout)
```

Description

`ddereq` requests data from a server application via an established DDE conversation. `ddereq` returns a matrix containing the requested data or an empty matrix if the function is unsuccessful.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddereq` returns a matrix containing the requested data in variable, `data`. Otherwise, it returns an empty matrix.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the server application's DDE item name for the data requested.
`format` (*optional*)	Two-element array specifying the format of the data requested. The first element specifies the Windows clipboard format to use. The only currently supported format is `cf_text`, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `numeric` (the default, which corresponds to `0`) and `string` (which corresponds to a value of `1`). The default format array is `[1 0]`.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Assume that we have an Excel spreadsheet `stocks.xls`. This spreadsheet contains the prices of three stocks in row 3 (columns 1 through 3) and the number of shares of these stocks in rows 6 through 8 (column 2). Initiate conversation with Excel with the command:

```
channel = ddeinit('excel','stocks.xls')
```

DDE functions require the *`rxcy`* reference style for Excel worksheets. In Excel terminology the prices are in `r3c1:r3c3` and the shares in `r6c2:r8c2`.

To request the prices from Excel:

```
prices = ddereq(channel,'r3c1:r3c3')

prices =
         42.50        15.00        78.88
```

To request the number of shares of each stock:

```
shares = ddereq(channel, 'r6c2:r8c2')

shares =
        100.00
        500.00
        300.00
```

See Also

ddeadv, ddeexec, ddeinit, ddepoke, ddeterm, ddeunadv

ddeterm

Purpose Terminate DDE conversation

Syntax

```
rc = ddeterm(channel)
```

Description `rc = ddeterm(channel)` accepts a channel handle returned by a previous call to `ddeinit` that established the DDE conversation. `ddeterm` terminates this conversation. `rc` is a return code where `0` indicates failure and `1` indicates success.

Examples To close a conversation channel previously opened with `ddeinit`:

```
rc = ddeterm(channel)

rc =

      1.00
```

See Also `ddeadv`, `ddeexec`, `ddeinit`, `ddepoke`, `ddereq`, `ddeunadv`

Purpose Release advisory link

Syntax

```
rc = ddeunadv(channel,'item')
rc = ddeunadv(channel,'item',format)
rc = ddeunadv(channel,'item',format,timeout)
```

Description `ddeunadv` releases the advisory link between MATLAB and the server application established by an earlier `ddeadv` call. The `channel`, *`item`*, and `format` must be the same as those specified in the call to `ddeadv` that initiated the link. If you include the `timeout` argument but accept the default `format`, you must specify `format` as an empty matrix.

If successful, `ddeunadv` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
`format` (*optional*)	Two-element array. This must be the same as the `format` argument for the corresponding `ddeadv` call.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Example To release an advisory link established previously with `ddeadv`:

```
rc = ddeunadv(channel, 'r1c1:r5c5')
rc =

     1.00
```

See Also `ddeadv`, `ddeexec`, `ddeinit`, `ddepoke`, `ddereq`, `ddeterm`

12

Serial Port I/O Functions

`clear (serial)`	Remove serial port object from MATLAB workspace
`delete (serial)`	Remove serial port object from memory
`disp (serial)`	Display serial port object summary information
`fclose (serial)`	Disconnect serial port object from the device
`fgetl (serial)`	Read from device and discard the terminator
`fgets (serial)`	Read from device and include the terminator
`fopen (serial)`	Connect serial port object to the device
`fprintf (serial)`	Write text to the device
`fread (serial)`	Read binary data from the device
`freeserial`	Release hold on a serial port
`fscanf (serial)`	Read data from device and format as text
`fwrite (serial)`	Write binary data to the device
`get (serial)`	Return serial port object properties
`instrcallback`	Display event information when an event occurs

`instrfind`	Return serial port objects from memory to the MATLAB workspace
`isvalid`	Determine if serial port objects are valid
`length (serial)`	Length of serial port object array
`load (serial)`	Load serial port objects and variables into MATLAB workspace
`readasync`	Read data asynchronously from the device
`record`	Record data and event information to a file
`save (serial)`	Save serial port objects and variables to MAT-file
`serial`	Create a serial port object
`serialbreak`	Send break to device connected to the serial port
`set (serial)`	Configure or display serial port object properties
`size (serial)`	Size of serial port object array
stopasync	Stop asynchronous read and write operations

Purpose Remove a serial port object from the MATLAB workspace

Syntax `clear obj`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `clear obj` removes `obj` from the MATLAB workspace.

Remarks If `obj` is connected to the device and it is cleared from the workspace, then `obj` remains connected to the device. You can restore `obj` to the workspace with the `instrfind` function. A serial port object connected to the device has a `Status` property value of `open`.

To disconnect `obj` from the device, use the `fclose` function. To remove `obj` from memory, use the `delete` function. You should remove invalid serial port objects from the workspace with `clear`.

If you use the `help` command to display help for `clear`, then you need to supply the pathname shown below.

```
help serial/private/clear
```

Example This example creates the serial port object `s`, copies `s` to a new variable `scopy`, and clears `s` from the MATLAB workspace. `s` is then restored to the workspace with `instrfind` and is shown to be identical to `scopy`.

```
s = serial('COM1');
scopy = s;
clear s
s = instrfind;
isequal(scopy,s)
ans =
     1
```

See Also

Functions

`delete`, `fclose`, `instrfind`, `isvalid`

Properties

Status

Purpose Remove a serial port object from memory

Syntax

```
delete(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `delete(obj)` removes `obj` from memory.

Remarks

When you delete `obj`, it becomes an *invalid* object. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command. If multiple references to `obj` exist in the workspace, then deleting one reference invalidates the remaining references.

If `obj` is connected to the device, it has a `Status` property value of `open`. If you issue `delete` while `obj` is connected, then the connection is automatically broken. You can also disconnect `obj` from the device with the `fclose` function.

If you use the `help` command to display help for `delete`, then you need to supply the pathname shown below.

```
help serial/delete
```

Example

This example creates the serial port object `s`, connects `s` to the device, writes and reads text data, disconnects `s` from the device, removes `s` from memory using `delete`, and then removes `s` from the workspace using `clear`.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

See Also

Functions

`clear`, `fclose`, `isvalid`

Properties

Status

disp (serial)

Purpose Display serial port object summary information

Syntax

```
obj
disp(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `obj` or `disp(obj)` displays summary information for `obj`.

Remarks In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when:

- Creating a serial port object
- Configuring property values using the dot notation

Use the display summary to quickly view the communication settings, communication state information, and information associated with read and write operations.

Example The following commands display summary information for the serial port object `s`.

```
s = serial('COM1')
s.BaudRate = 300
s
```

fclose (serial)

Purpose Disconnect a serial port object from the device

Syntax `fclose(obj)`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `fclose(obj)` disconnects `obj` from the device.

Remarks If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the device using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

If you use the `help` command to display help for `fclose`, then you need to supply the pathname shown below.

```
help serial/fclose
```

Example This example creates the serial port object `s`, connects `s` to the device, writes and reads text data, and then disconnects `s` from the device using `fclose`.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
```

At this point, the device is available to be connected to a serial port object. If you no longer need `s`, you should remove from memory with the `delete` function, and remove it from the workspace with the `clear` command.

See Also

Functions

`clear`, `delete`, `fopen`, `stopasync`

Properties

`RecordStatus`, `Status`

fgetl (serial)

Purpose Read one line of text from the device and discard the terminator

Syntax

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

Arguments

`obj`	A serial port object.
`tline`	Text read from the instrument, excluding the terminator.
`count`	The number of values read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`tline = fgetl(obj)` reads one line of text from the device connected to `obj`, and returns the data to `tline`. The returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Remarks

Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgetl` is issued.

If you use the `help` command to display help for `fgetl`, then you need to supply the pathname shown below.

```
help serial/fgetl
```

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    17
```

Use `fgetl` to read the data returned from the previous write operation, and discard the terminator.

```
settings = fgetl(s)
settings =
9600;0;0;NONE;LF
length(settings)
ans =
    16
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

`fgets`, `fopen`

Properties

`BytesAvailable`, `InputBufferSize`, `ReadAsyncMode`, `Status`, `Terminator`, `Timeout`, `ValuesReceived`

Purpose Read one line of text from the device and include the terminator

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
```

Arguments

`obj`	A serial port object.
`tline`	Text read from the instrument, including the terminator.
`count`	The number of bytes read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`tline = fgets(obj)` reads one line of text from the device connected to `obj`, and returns the data to `tline`. The returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Remarks

Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgets` is issued.

If you use the `help` command to display help for `fgets`, then you need to supply the pathname shown below.

```
help serial/fgets
```

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    17
```

Use `fgets` to read the data returned from the previous write operation, and include the terminator.

```
settings = fgets(s)
settings =
9600;0;0;NONE;LF
length(settings)
ans =
    17
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

`fgetl`, `fopen`

Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, Timeout, ValuesReceived

fopen (serial)

Purpose Connect a serial port object to the device

Syntax `fopen(obj)`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `fopen(obj)` connects `obj` to the device.

Remarks Before you can perform a read or write operation, `obj` must be connected to the device with the `fopen` function. When `obj` is connected to the device:

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the device. You can connect only one serial port object to a given device.

Some properties are read-only while the serial port object is open (connected), and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the device. If any of these properties are incorrectly configured, then an error is returned when `fopen` is issued and `obj` is not connected to the device. Properties of this type include `BaudRate`, and are associated with device settings.

If you use the `help` command to display help for `fopen`, then you need to supply the pathname shown below.

```
help serial/fopen
```

Example This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
```

See Also

Functions

fclose

Properties

`BytesAvailable`, `BytesToOutput`, `Status`, `ValuesReceived`, `ValuesSent`

fprintf (serial)

Purpose Write text to the device

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Arguments

`obj`	A serial port object.
`'cmd'`	The string written to the device.
`'format'`	C language conversion specification.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution is complete.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sprintf` file I/O format specifications or a C manual for more information.

`fprintf(obj,'cmd','mode')` writes the string with command line access specified by *`mode`*. If *`mode`* is `sync`, `cmd` is written synchronously and the command line is blocked. If *`mode`* is `async`, `cmd` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. If *`mode`* is `sync`, `cmd` is written synchronously. If *`mode`* is `async`, `cmd` is written asynchronously.

Remarks

Before you can write text to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fprintf` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you use the `help` command to display help for `fprintf`, then you need to supply the pathname shown below.

```
help serial/fprintf
```

Synchronous Versus Asynchronous Write Operations

By default, text is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Controlling Access to the MATLAB Command Line.

Rules for Completing a Write Operation with fprintf

A synchronous or asynchronous write operation using `fprintf` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Additionally, you can stop an asynchronous write operation with the `stopasync` function.

Rules for Writing the Terminator

All occurrences of `\n` in `cmd` are replaced with the `Terminator` property value. Therefore, when using the default format `%s\n`, all commands written to the device will end with this property value. The terminator required by your device will be described in its documentation.

Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default format for `fprintf` is `%s\n`, the terminator specified by the `Terminator` property was automatically written. However, in some cases you might want to suppress writing the terminator. To do so, you must explicitly specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
fprintf(s,'%s','RS232?')
```

See Also

Functions

`fopen`, `fwrite`, `stopasync`

Properties

`BytesToOutput`, `OutputBufferSize`, `OutputEmptyFcn`, `Status`, `TransferStatus`, `ValuesSent`

Purpose

Read binary data from the device

Syntax

```
A = fread(obj,size)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Arguments

`obj`	A serial port object.
`size`	The number of values to read.
`'precision'`	The number of bits read for each value, and the interpretation of the bits as character, integer, or floating-point values.
`A`	Binary data returned from the device.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = fread(obj,size)` reads binary data from the device connected to `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

`A = fread(obj,size,'precision')` reads binary data with precision specified by *`precision`*.

precision controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *precision* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *precision* are listed below in Remarks.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

Remarks

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

If you use the `help` command to display help for `fread`, then you need to supply the pathname shown below.

```
help serial/fread
```

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB command line until:

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used for binary read operations.

Supported Precisions

The supported values for *precision* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

Functions

fgetl, fgets, fopen, fscanf

Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, ValuesReceived

Purpose Release hold on a serial port

Syntax

```
freeserial
freeserial('port')
freeserial(obj)
```

Arguments

`'port'`	A serial port name, or a cell array of serial port names
`obj`	A serial port object, or an array of serial port objects.

Description

`freeserial` releases the hold MATLAB has on all serial ports.

`freeserial('port')` releases the hold MATLAB has on the serial port specified by `port`. `port` can be a cell array of strings.

`freeserial(obj)` releases the hold MATLAB has on the serial port associated with the object specified by `obj`. `obj` can be an array of serial port objects.

Remarks

An error is returned if a serial port object is connected to the port that is being freed. Use the `fclose` function to disconnect the serial port object from the serial port.

`freeserial` is necessary only on Windows platforms. You should use `freeserial` if you need to connect to the serial port from another application after a serial port object has been connected to that port, and you do not want to exit MATLAB.

See Also

Functions

fclose

fscanf (serial)

Purpose

Read data from the device, and format as text

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Arguments

`obj`	A serial port object.
`'format'`	C language conversion specification.
`size`	The number of values to read.
`A`	Data read from the device and formatted as text.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = fscanf(obj)` reads data from the device connected to `obj`, and returns it to `A`. The data is converted to text using the `%c` format.

`A = fscanf(obj,'format')` reads data and converts it according to *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. If `size` is not of the form `[m,n]`, and a character conversion is specified, then `A` is returned as a row vector. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. An ASCII value is one byte.

`[A,count] = fscanf(...)` returns the number of values read to `count`.

`[A,count,msg] = fscanf(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Remarks

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fscanf` is issued.

If you use the `help` command to display help for `fscanf`, then you need to supply the pathname shown below.

```
help serial/fscanf
```

Rules for Completing a Read Operation with fscanf

A read operation with `fscanf` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The number of values specified by `size` is read.
- The input buffer is filled (unless `size` is specified)

Example

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying sine wave.

```
s = serial('COM1');
fopen(s)
```

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',14)
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

`fgetl`, `fgets`, `fopen`, `fread`, `strread`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `InputBufferSize`, `Status`, `Terminator`, `Timeout`

Purpose Write binary data to the device

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A, mode')
fwrite(obj,A,'precision', mode')
```

Arguments

`obj`	A serial port object.
`A`	The binary data written to the device.
`'precision'`	The number of bits written for each value, and the interpretation of the bits as character, integer, or floating-point values.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description `fwrite(obj,A)` writes the binary data `A` to the device connected to `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *`precision`*.

`precision` controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. The supported values for *`precision`* are listed below in Remarks.

`fwrite(obj,A,'mode')` writes binary data with command line access specified by *`mode`*. If *`mode`* is `sync`, `A` is written synchronously and the command line is blocked. If *`mode`* is `async`, `A` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *`precision`* and command line access specified by *`mode`*.

Remarks Before you can write data to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you use the `help` command to display help for `fwrite`, then you need to supply the pathname shown below.

```
help serial/fwrite
```

Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Writing Data.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used with binary write operations.

Supported Precisions

The supported values for *precision* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

Functions

`fopen`, `fprintf`

Properties

`BytesToOutput`, `OutputBufferSize`, `OutputEmptyFcn`, `Status`, `Timeout`, `TransferStatus`, `ValuesSent`

Purpose Return serial port object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`'PropertyName'`	A property name or a cell array of property names.
`out`	A single property value, a structure of property values, or a cell array of property values.

Description

`get(obj)` returns all property names and their current values to the command line for `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-n or n-by-1 cell array of strings containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of serial port objects, then `out` will be a m-by-n cell array of property values where m is equal to the length of `obj` and n is equal to the number of properties specified.

Remarks

Refer to "Displaying Property Names and Property Values" for a list of serial port object properties that you can return with `get`.

When you specify a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then these commands are all valid.

```
out = get(s,'BaudRate');
out = get(s,'baudrate');
out = get(s,'BAUD');
```

If you use the help command to display help for get, then you need to supply the pathname shown below.

```
help serial/get
```

Example

This example illustrates some of the ways you can use get to return property values for the serial port object s.

```
s = serial('COM1');
out1 = get(s);
out2 = get(s,{'BaudRate','DataBits'});
get(s,'Parity')
ans =
none
```

See Also

Functions

set

Purpose Display event information when an event occurs

Syntax `instrcallback(obj,event)`

Arguments

`obj`	An serial port object.
`event`	The event that caused the callback to execute.

Description `instrcallback(obj,event)` displays a message that contains the event type, the time the event occurred, and the name of the serial port object that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.

Remarks You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

Example The following example creates the serial port objects `s`, and configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

instrfind

Purpose

Return serial port objects from memory to the MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Arguments

`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *`PropertyName`*.
`S`	A structure of property names and property values.
`obj`	A serial port object, or an array of serial port objects.
`out`	An array of serial port objects.

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('PropertyName',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'PropertyName',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Remarks

Refer to "Displaying Property Names and Property Values" for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

Example

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind

   Instrument Object Array
   Index:   Type:          Status:     Name:
   1        serial         open        Serial-COM1
   2        serial         open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

See Also

Functions

`clear`, `get`

isvalid

Purpose Determine if serial port objects are valid

Syntax

```
out = isvalid(obj)
```

Arguments

`obj`	A serial port object or array of serial port objects.
`out`	A logical array.

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a `0` where the elements of `obj` are invalid serial port objects and a `1` where the elements of `obj` are valid serial port objects.

Remarks

`obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command.

Example

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM1');
```

`s2` becomes invalid after it is deleted.

```
delete(s2)
```

`isvalid` verifies that `s1` is valid and `s2` is invalid.

```
sarray = [s1 s2];
isvalid(sarray)
ans =
     1     0
```

See Also

Functions

`clear`, `delete`

Purpose Length of serial port object array

Syntax `length(obj)`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `length(obj)` returns the length of `obj`. It is equivalent to the command `max(size(obj))`.

See Also

Functions

size

load (serial)

Purpose

Load serial port objects and variables into the MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2...
out = load('filename','obj1','obj2',...)
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2...`	Serial port objects or arrays of serial port objects.
`out`	A structure containing the specified serial port objects.

Description

`load filename` returns all variables from the MAT-file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2...` returns the serial port objects specified by `obj1 obj2` ... from the MAT-file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified serial port objects from the MAT-file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded serial port objects.

Remarks

Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

If you use the `help` command to display help for `load`, then you need to supply the pathname shown below.

```
help serial/private/load
```

Example

Suppose you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments.

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s1,'Parity','mark','DataBits',7)
fopen(s1)
```

```
fopen(s2)
```

Save `s1` and `s2` to the file `MyObject.mat`, and then load the objects into the workspace using new variables.

```
save MyObject s1 s2
news1 = load MyObject s1
news2 = load('MyObject','s2')
```

Values for read-only properties are restored to their default values upon loading, while all other properties values are honored.

```
get(news1,{'Parity','DataBits','Status'})
ans =
    'mark'    [7]    'closed'
get(news2,{'Parity','DataBits','Status'})
ans =
    'none'    [8]    'closed'
```

See Also

Functions

save

Properties

Status

readasync

Purpose Read data asynchronously from the device

Syntax

```
readasync(obj)
readasync(obj,size)
```

Arguments

`obj`	A serial port object.
`size`	The number of bytes to read from the device.

Description `readasync(obj)` initiates an asynchronous read operation.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Remarks Before you can read data, you must connect `obj` to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

You should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute an M-file callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if `size` is not specified).

Because `readasync` checks for the terminator, this function can be slow. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

Example

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
ans =
     15
out = fscanf(s)
out =
2.0399999619E0
fclose(s)
```

See Also

Functions

`fopen`, `stopasync`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `ReadAsyncMode`, `Status`, `TransferStatus`

Purpose Record data and event information to a file

Syntax

```
record(obj)
record(obj,'switch')
```

Arguments

`obj`	A serial port object.
`'switch'`	Switch recording capabilities on or off.

Description

`record(obj)` toggles the recording state for `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Remarks

Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to "Debugging: Recording Information to Disk."

Example

This example creates the serial port object `s`, connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
s.RecordName = 'MySerialFile.txt';
record(s,'on')
```

```
fprintf(s,'*IDN?')
out = fscanf(s);
record(s,'off')
fclose(s)
```

See Also

Functions

`fclose`, `fopen`

Properties

`RecordDetail`, `RecordMode`, `RecordName`, `RecordStatus`, `Status`

Purpose Save serial port objects and variables to a MAT-file

Syntax

```
save filename
save filename obj1 obj2...
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2...`	Serial port objects or arrays of serial port objects.

Description

`save filename` saves all MATLAB variables to the MAT-file `filename`. If an extension is not specified for `filename`, then the `.mat` extension is used.

`save filename obj1 obj2...` saves the serial port objects `obj1 obj2` ... to the MAT-file `filename`.

Remarks

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example. to save the serial port object `s` to the file `MySerial.mat`

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for `obj`. To save that data to a MAT-file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the MAT-file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

If you use the `help` command to display help for `save`, then you need to supply the pathname shown below.

```
help serial/private/save
```

Example

This example illustrates how to use the command and functional form of `save`.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

See Also

Functions

`load`, `record`

Properties

Status

Purpose

Create a serial port object

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

Arguments

`'port'`	The serial port name.
`'PropertyName'`	A serial port property name.
`PropertyValue`	A property value supported by *PropertyName*.
`obj`	The serial port object.

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by `port`. If `port` does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`obj = serial('port','PropertyName',PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Remarks

When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid.

```
s = serial('COM1','BaudRate',4800);
```

```
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to “Configuring Property Values” for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

Example

This example creates the serial port object `s1` associated with the serial port COM1.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

See Also

Functions

`fclose`, `fopen`

Properties

`Name`, `Port`, `Status`, `Type`

serialbreak

Purpose

Send a break to the device connected to the serial port

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Arguments

`obj`	A serial port object.
`time`	The duration of the break, in milliseconds.

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the device connected to `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

Remarks

For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the device.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also

Functions

`fopen`, `stopasync`

Properties

Status

set (serial)

Purpose

Configure or display serial port object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *`PropertyName`*.
`PN`	A cell array of property names.
`PV`	A cell array of property values.
`S`	A structure with property names and property values.
`props`	A structure array whose field names are the property names for `obj`, or cell array of possible values.

Description

`set(obj)` displays all configurable properties values for `obj`. If a property has a finite list of possible string values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for `obj` to `props`. `props` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj,'PropertyName')` displays the valid values for *`PropertyName`* if it possesses a finite list of string values.

`props = set(obj,'PropertyName')` returns the valid values for *`PropertyName`* to `props`. `props` is a cell array of possible string values or an empty cell array if *`PropertyName`* does not have a finite list of possible values.

`set(obj,'PropertyName',PropertyValue,...)` configures multiple property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be m-by-n where m is equal to the number of serial port objects in `obj` and n is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.

Remarks

Refer to "Configuring Property Values" for a list of serial port object properties that you can configure with `set`.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then the following commands are all valid.

```
set(s,'BaudRate')
set(s,'baudrate')
set(s,'BAUD')
```

If you use the `help` command to display help for `set`, then you need to supply the pathname shown below.

```
help serial/set
```

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the serial port object `s`.

```
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},{2,'sydney.txt'})
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

See Also

Functions

get

Purpose

Size of serial port object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`dim`	The dimension of `obj`.
`d`	The number of rows and columns in `obj`.
`m`	The number of rows in `obj`, or the length of the dimension specified by `dim`.
`n`	The number of columns in `obj`.
`m1,m2,..., mn`	The length of the first N dimensions of `obj`.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

Functions

length

stopasync

Purpose Stop asynchronous read and write operations

Syntax `stopasync(obj)`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `stopasync(obj)` stops any asynchronous read or write operation that is in progress for `obj`.

Remarks You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

`ReadAsyncMode`, `TransferStatus`