MATLAB®

C/C++, Fortran, Java, and Python API Reference

MATLAB®

R2017b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

MATLAB® C/C++, Fortran, Java®, and Python® API Reference

Revision History

December 1996	First Printing	New for MATLAB 5 (Release 8)
May 1997	Online only	Revised for MATLAB 5.1 (Release 9)
January 1998	Online Only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online Only	Revised for MATLAB 5.3 (Release 11)
September 2000	Online Only	Revised for MATLAB 6.0 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
January 2003	Online only	Revised for MATLAB 6.5.1 (Release 13SP1)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised and renamed for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised and renamed for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised and renamed for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised and renamed for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised and renamed for MATLAB 7.10 (Release 2010a)
September 2010	Online only	Revised for MATLAB 7.11 (Release 2010b)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online only	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Online only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online only	Revised for MATLAB 9.3 (Release 2017b)

Contents

1 API Reference

API Reference

matlab::data::ArrayDimensions

Type specifying array dimensions

Description

Use the `ArrayDimensions` type to specify the size of an array. `ArrayDimensions` is specified as:

```
using ArrayDimensions = std::vector<size_t>;
```

Free Function

getNumElements

```
inline size_t getNumElements(const ArrayDimensions& dims)
```

Determine the number of elements based on the `ArrayDimensions`.

`const ArrayDimensions& dims`	Array dimensions.

`inline size_t`	Number of elements.

None

See Also

Topics

"MATLAB Data API Types"

Introduced in R2017b

matlab::data::ArrayElementRef

C++ class representing element of Array or Reference<Array> object

Description

An `ArrayElementRef` object is created when using `operator[]` into an `Array` or a `Reference<Array>`. It is untyped since `Array` and `Reference<Array>` do not have type information. It collects up the indexes specified by the user and can be cast to an element if the array holds primitive data or can be used to create a `Reference<T>` instance.

Indexing is zero-based.

Class Details

Namespace:	`matlab::data`
Include:	`ArrayElementRef.hpp`

Template Parameters

`bool is_const_ref`	Indicates if this reference is `const`. The non-`const` version of this class supports assignment.

Other Operators

- "operator[]" on page 1-4
- "operator T" on page 1-5
- "operator=" on page 1-6

`operator[]`

```
ArrayElementRef<is_const_ref> operator[](size_t idx)
```

```
ArrayElementRef<is_const_ref> operator[](std::string idx)
```

Add an index to an Array.

`size_t idx`	Index to add. Call this syntax to use two or more indices in an indexing operation.
`std::string idx`	Index to add. Array must support string indexing.

`TooManyIndicesProvidedException`	Too many indices provided. The number of `size_t` indices is more than the number of dimensions in the array.
`StringIndexMustBeLastException`	String index is not the last index. A `size_t` index is provided after a string index.
`StringIndexNotValidException`	An `std::string` index is not valid for this array.
`CanOnlyUseOneStringIndexException`	More than one `std::string` index provided.

```
double val = arr[2][7]["f1"];
```

operator T

```
template <typename T>
operator T() const
```

Cast a value. Throws an error if not arithmetic, complex, or `std::string`.

`InvalidArrayIndexException`	The index provided is not valid for this Array or one of the indices is out of range.
`TypeMismatchException`	The element of the Array does not contain the type specified by `T`.

```
Array dblArray_const = factory.createArray<double>({2,2});
double val = dblArray_const[1][2];
```

operator=

```
template <typename T>
ArrayElementRef<is_const_ref>& operator= (T rhs)
```

Assign a primitive or complex type to an element of an Array. The Array must be non-`const`.

If the Array contains other Arrays, then `operator=` can assign an Array into the element.

`T rhs`	The value to assign to an element in the Array, specified as a primitive or complex type or as an Array.

`ArrayElementRef<is_const_ref>&`	Reference to the Array element.

`InvalidArrayIndexException`	The index provided is not valid for this Array or one of the indices is out of range.
`TypeMismatchException`	The element of the Array does not contain a primitive or complex type or does not contain other Arrays.

Assign numeric types to Array `arr`.

```
Array arr = factory.createArray<double>({2,2});
arr[0][0] = 5.5;
arr[1][2] = std::complex<double>(5.4, 3.1);
```

Assign an Array to `cell_arr`, then create a `Reference<Array> ref` to the element.

```
Array cell_arr = factory.createArray<Array>({1,3});
cell_arr[2] = factory.createScalar(true);
Reference<Array> ref = cell_arr[2];
```

See Also

`Array` | `ArrayElementTypedRef` | `Reference` | `TypedArrayRef`

Introduced in R2017b

matlab::data::ArrayElementTypedRef

C++ class representing element of TypedArray or TypedArrayRef object

Description

An `ArrayElementTypedRef` object is created when using `operator[]` into a `TypedArray<T>` or a `TypedArrayRef<T>`. It collects up the indexes specified by the user and can be cast to `<T>` without the added cost of type checking that exists when using `ArrayElementRef`. Primitive types can be cast to T& which is not supported with `ArrayElementRef`. The non-`const` version of this class supports assignment.

Class Details

Namespace:	`matlab::data`
Include:	`ArrayElementTypedRef.hpp`

Template Parameters

`T`	Element types.
`bool is_const_ref`	TBD

Other Operators

- "operator[]" on page 1-8
- "operator reference" on page 1-9
- "operator=" on page 1-10

operator[]

```
ArrayElementTypedRef<T, is_const_ref> operator[](size_t idx)
```

```
ArrayElementTypedRef<Array, is_const_ref> operator[](std::string
idx)
```

Add an index to an Array.

`size_t idx`	Index to add. Call this syntax to use two or more indices in an indexing operation.
`std::string idx`	Index to add. Array must support string indexing.

`ArrayElementTypedRef<Array, is_const_ref>`	New instance with the additional index.

`TooManyIndicesProvidedException`	Too many indices provided.
`StringIndexMustBeLastException`	String index is not the last index.

```
double val = arr[2][7][3];
```

operator reference

```
operator reference() const
```

```
operator std::string() const
```

Cast to reference to its primitive value.

`T`	The value.
`std::string`	

`InvalidArrayIndexException`	The index provided is not valid for this Array or one of the indices is out of range.
`TypeMismatchException`	The element of the Array cannot be cast to `std::string`.

```
double val = dblArray[1][2];
```

```
std::string val = strArray[1][2];
```

operator=

```
ArrayElementTypedRef<T, is_const_ref>& operator= (T rhs)
```

```
ArrayElementTypedRef<T, is_const_ref>& operator= (std::string rhs)
```

Assign a value into an Array. The Array must be non-`const`.

`T rhs`
 	
`std::string rhs`
	Value to assign to an element in the Array

`ArrayElementTypedRef<T, is_const_ref>&`	Reference to the Array element.

`InvalidArrayIndexException`	The index provided is not valid for this Array or one of the indices is out of range.
`TypeMismatchException`	The element of the Array cannot be cast to `std::string`.

```
arr[1][2] = std::complex<double>(5.4, 3.1);
```

```
arr[1][2] = "MyString";
```

Examples

TBD

```
TypedArray<double> arr = factory.createArray<double>({2,2});
arr[0][0] = 5.5;
double& val = arr[0][0];
CellArray cell_arr = factory.createArray<Array>({1,2});
cell_arr[0] = factory.createScalar(10.5);
cell_arr[1] = factory.createScalar(false);
ArrayRef ref_to_element = cell_arr[0];
TypedArrayRef<double> typed_ref_to_element = cell_arr[0];
```

```
Array const shared_copy_of_element = cell_arr[1];
TypedArray<bool> const typed_shared_copy_of_element = cell_arr[1];
```

See Also

`Array` | `ArrayElementRef` | `ArrayElementTypedRef` | `Reference` | `TypedArrayRef` | `TypedIterator`

Introduced in R2017b

matlab::data::ArrayFactory

C++ class to create arrays

Description

Use `ArrayFactory` to create `matlab::data::Array` objects.

Class Details

Namespace:	`matlab::data`
Include:	`ArrayFactory.hpp`

Constructors

Default Constructor

```
ArrayFactory()
```

`FailedToLoadLibMatlabDataArrayException`	Concrete implementation has not been loaded.

Destructor

```
~ArrayFactory()
```

Member Functions

- "createArray" on page 1-13
- "createScalar" on page 1-14
- "createCellArray" on page 1-15

- “createCharArray” on page 1-16
- “createStructArray” on page 1-17
- “createEnumArray” on page 1-18
- “createSparseArray” on page 1-19
- “createEmptyArray” on page 1-20
- “createBuffer” on page 1-21
- “createArrayFromBuffer” on page 1-21

createArray

```
template <typename T>
TypedArray<T> createArray(ArrayDimensions dims)

template <typename ItType, typename T>
TypedArray<T> createArray(ArrayDimensions dims,
    ItType begin,
    ItType end)

template <typename T>
TypedArray<T> createArray(ArrayDimensions dims,
    const T* const begin,
    const T* const end)

template <typename T>
TypedArray<T> createArray(ArrayDimensions dims,
    std::initializer_list<T> data)
```

Creates a `TypedArray<T>` with the given dimensions. Fills the array with data, if specified. Data is copied and must be in column major order.

- `ItType` - Iterator types, specified as `std::iterator`.
- `T` - Element types, specified as one of the following C++ data types.

bool	int8_t	int16_t	int32_t	int64_t	uint8_t
uint16_t	uint32_t	uint64_t	float	double	char16_t

matlab::data::String	std::complex<double>	std::complex<float>	std::complex<int8_t>	std::complex<uint8_t>	std::complex<int16_t>
std::complex<uint16_t>	std::complex<int32_t>	std::complex<uint32_t>	std::complex<int64_t>	std::complex<uint64_t>	matlab::data::MATLABString

`ArrayDimensions dims`	Dimensions for the array.
`ItType begin` `ItType end`	Start and end of the user supplied data. The data type is determined by the `value_type` of the iterator.
`const T* const begin` `const T* const end`	Start and end of the user supplied data specified as C-style pointer. Supports all primitive types, complex types, and string types.
`std::initializer_list<T>`	Initializer list containing the data.

`matlab::OutOfMemoryException`	The array could not be allocated.

createScalar

```
template <typename T>
TypedArray<T> createScalar(const T val)

TypedArray<String> createScalar(const String val)

TypedArray<String> createScalar(const std::string val)
```

Creates a scalar `TypedArray<T>` with the given value. This method supports arithmetic types, complex types, and string types.

`const T val`
`const String val`
`const std::string val`
	Value to be inserted into the scalar. If `val` is 7-bit ASCII data, then the method converts it to UTF16.

matlab::OutOfMemoryException	The array could not be allocated.
NonAsciiCharInInputDataException	Input is `std::string` and contains non-ASCII characters.

```
#include "MatlabDataArray.hpp"

int main() {
    matlab::data::ArrayFactory factory;

    // Create a vector containing 2 scalar values
    std::vector<matlab::data::Array> args({
    factory.createScalar<int16_t>(100),
    factory.createScalar<int16_t>(60)});
    return 0;
}
```

"Call Function with Single Returned Argument"

createCellArray

```
CellArray<Array> createCellArray(ArrayDimensions dims)

template <typename ...Targs>
CellArray<Array> createCellArray(ArrayDimensions dims, Targs... data)
```

Creates a `CellArray` with the specified `data`. Data is specified in column major order.

...Targs	Variadic template of: • arithmetic type • complex type • `matlab::data::String` • `std::string` • `matlab::data::Array`

`ArrayDimensions dims`	Dimensions of the cell array.
`Targs... data`	Elements to be inserted into the cell array, specified as a primitive complex type, string, or `Array`.

`matlab::OutOfMemoryException`	The array could not be allocated.
`NonAsciiCharInInputDataException`	Input is `std::string` and contains non-ASCII characters.

```
#include "MatlabDataArray.hpp"

int main()
{
    using namespace matlab::data;
    ArrayFactory f;
    CellArray myArray = f.createCellArray({ 1,2 },
        f.createCharArray("MATLAB Cell Array"),
        f.createArray<double>({ 2,2 }, { 1.2, 2.2, 3.2, 4.2 }));

    return 0;
}
```

createCharArray

```
CharArray createCharArray(String str)

CharArray createCharArray(std::string str)
```

Creates a 1xn `CharArray` from the specified input, where n is the string length.

`matlab::data::String str`	Data to be filled into the array.
`std::string str`	

`matlab::OutOfMemoryException`	The array could not be allocated.

NonAsciiCharInInputDataException	Input is `std::string` and contains non-ASCII characters.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
    return 0;
}
```

createStructArray

```
StructArray createStructArray(ArrayDimensions dims,
    std::vector<std::string> fieldNames)
```

Creates a `StructArray` with the given dimensions and field names.

`ArrayDimensions dims`	Dimensions for the array.
`std::vector<std::string> fieldNames`	Vector of the field names for the structure.

matlab::OutOfMemoryException	The array could not be allocated.
DuplicateFieldNameInStructArrayException	Duplicate field names specified.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory f;

    // Create StructArray equivalent to MATLAB structure s:
    // s = struct('loc', {'east', 'west'}, 'data', {[1, 2, 3], [4., 5., 6., 7., 8.]})
    StructArray S = f.createStructArray({ 1,2 }, { "loc", "data" });
    S[0]["loc"] = f.createCharArray("east");
    S[0]["data"] = f.createArray<uint8_t>({ 1, 3 }, { 1, 2, 3 });
```

```
    S[1]["loc"] = f.createCharArray("west");
    S[1]["data"] = f.createArray<double>({ 1, 5 }, { 4., 5., 6., 7., 8. });

    // Access the value defined by the MATLAB statement:
    // s(1).data
    Reference<Array> val = S[0]["data"];

    return 0;
}
```

createEnumArray

```
EnumArray createEnumArray(ArrayDimensions dims,
    std::string className,
    std::vector<std::string> enums)

EnumArray createEnumArray(ArrayDimensions dims,
    std::string className)
```

Creates an `EnumArray` of type `className`. If specified, the array is initialized with the list of enumeration names.

`ArrayDimensions dims`	Dimensions for the array.
`std::string className`	Class name of the enumeration array.
`std::vector<std::string> enums`	List of the enumeration names.

`matlab::OutOfMemoryException`	The array could not be allocated.
`MustSpecifyClassNameException`	Class name not specified.
`WrongNumberOfEnumsSuppliedException`	The wrong number of enumerations provided.

```
#include "MatlabDataArray.hpp"
#include <vector>
```

```
int main()
{
    using namespace matlab::data;
    ArrayFactory f;
    auto blue = f.createEnumArray({ 1,1 }, "TextColor", { "Blue" });

    // Create an argument vector
    std::vector<Array> args({ f.createCharArray("My text"), std::move(blue) });

    return 0;
}
```

createSparseArray

```
template <typename T>
SparseArray<T> createSparseArray(ArrayDimensions dims,
    size_t nnz,
    buffer_ptr_t<T> data,
    buffer_ptr_t<size_t> rows,
    buffer_ptr_t<size_t> cols)
```

Creates a `SparseArray<T>` with `rows`-by-`cols` dimensions. Only two dimensions are allowed for sparse arrays. The buffer is not copied and the array takes ownership.

`T`	Element types, specified as `double`, `bool`, or `std::complex<double>`.

`ArrayDimensions dims`	Dimensions for the array.
`size_t nnz`	Number of nonzero elements.
`buffer_ptr_t<T> data`	Buffer containing the nonzero elements.
`buffer_ptr_t<size_t> rows`	Buffer containing the row value for each element.
`buffer_ptr_t<size_t> cols`	Buffer containing the column value for each element.

matlab::OutOfMemoryException	The array could not be allocated.
InvalidDimensionsInSparseArrayException	More than two dimensions are specified.

```
#include "MatlabDataArray.hpp"

int main() {
    std::vector<double> data = { 3.5, 12.98, 21.76 };
    std::vector<size_t> rows = { 0,0,1 };
    std::vector<size_t> cols = { 0,4,8 };
    size_t nnz = 3;

    matlab::data::ArrayFactory factory;
    auto data_p = factory.createBuffer<double>(nnz);
    auto rows_p = factory.createBuffer<size_t>(nnz);
    auto cols_p = factory.createBuffer<size_t>(nnz);

    double* dataPtr = data_p.get();
    size_t* rowsPtr = rows_p.get();
    size_t* colsPtr = cols_p.get();
    std::for_each(data.begin(), data.end(), [&](const double& e) { *(dataPtr++) = e; })
    std::for_each(rows.begin(), rows.end(), [&](const size_t& e) { *(rowsPtr++) = e; })
    std::for_each(cols.begin(), cols.end(), [&](const size_t& e) { *(colsPtr++) = e; })

    matlab::data::SparseArray<double> arr =
        factory.createSparseArray<double>({ 2,9 }, nnz, std::move(data_p),
            std::move(rows_p), std::move(cols_p));
    return 0;
}
```

createEmptyArray

```
Array createEmptyArray()
```

Creates an empty `Array` containing no elements.

Array	Empty array.

matlab::OutOfMemoryException	The array could not be allocated.

createBuffer

```
template <typename T>
buffer_ptr_t<T> createBuffer(size_t numberOfElements)
```

Creates a buffer which can be passed into `createArrayFromBuffer`. No data copies are made when creating an array from a buffer. Data must be in column major order.

T	Primitive types

size_t numberOfElements	Number of elements, not the actual buffer size.

buffer_ptr_t<T>	Unique_ptr containing the buffer.

matlab::OutOfMemoryException	The array could not be allocated.

createArrayFromBuffer

```
template <typename T>
TypedArray<T> createArrayFromBuffer(ArrayDimensions dims,
    buffer_ptr_t<T> buffer)
```

Creates a `TypedArray<T>` using the given buffer.

T	Primitive types

ArrayDimensions dims	Dimensions for the array.
buffer_ptr_t<T> buffer	Buffer containing the data. The buffer is not copied. The `TypedArray<T>` object takes ownership of the buffer. Data must be in column major order.

`matlab::OutOfMemoryException`	The array could not be allocated.

See Also

Introduced in R2017b

matlab::data::Reference<Array>

C++ class to get reference to `Array`

Description

Use the `Reference<Array>` class to get a reference to an `Array` element of a container object, such as a MATLAB structure or cell array. The class is a base class for all reference types that refer to arrays and provides basic array information. `ArrayRef` is defined as:

```
using ArrayRef = Reference<Array>;
```

Class Details

Namespace:	`matlab::data`
Include:	`ArrayReferenceExt.hpp`

Member Functions

- “getType” on page 1-23
- “getDimensions” on page 1-24
- “getNumberOfElements” on page 1-24
- “isEmpty” on page 1-24

getType

```
ArrayType getType() const
```

`ArrayType`	Type of the array

`NotEnoughIndicesProvidedException`	Not enough indices provided.

InvalidArrayIndexException	Index provided is not valid for this Array or one of the indices is out of range.
InvalidArrayTypeException	Array type not recognized.

getDimensions

```
ArrayDimensions getDimensions() const
```

ArrayDimensions	Array dimensions vector.

NotEnoughIndicesProvidedException	Not enough indices provided.
InvalidArrayIndexException	Index provided is not valid for this Array or one of the indices is out of range.

getNumberOfElements

```
size_t getNumberOfElements() const
```

size_t	Number of elements in array.

NotEnoughIndicesProvidedException	Not enough indices provided.
InvalidArrayIndexException	Index provided is not valid for this Array or one of the indices is out of range.

isEmpty

```
bool isEmpty() const
```

bool	Returns true if array is empty, otherwise returns false.

NotEnoughIndicesProvidedException	Not enough indices provided.

`InvalidArrayIndexException`	Index provided is not valid for this `Array` or one of the indices is out of range.

Free Functions

- "getReadOnlyElements" on page 1-25
- "getWritableElements" on page 1-25

getReadOnlyElements

```
template <typename T>
Range<TypedIterator, T const> getReadOnlyElements(const Reference<Array>& ref)
```

Get a range containing the elements of the `Array` or `Reference<Array>`. Iterators contained in the range are `const`.

`const Reference<Array>& ref`	`Reference<Array>`.

`Range<TypedIterator, T const>`	Range containing `begin` and `end` iterators for the elements of the input `Reference<Array>`.

`InvalidArrayTypeException`	`Array` does not contain type `T`.

getWritableElements

```
template <typename T>
Range<TypedIterator, T> getWritableElements(Reference<Array>& ref)
```

Get a range containing the elements of the `Array` or `Reference<Array>`. Iterators contained in the range are non-`const`.

`Reference<Array>& ref`	`Reference<Array>`.

`Range<TypedIterator, T>`	Range containing `begin` and `end` iterators for the elements of the input `Reference<Array>`.

`InvalidArrayTypeException`	`Array` does not contain type `T`.

See Also

`ArrayType`

Introduced in R2017b

matlab::data::ArrayType

C++ array type enumeration class

Description

Use `ArrayType` objects to identify the data type and other attributes of a MATLAB array.

Class Details

Namespace:	`matlab::data`
Include:	`ArrayType.hpp`

Enumeration

```
enum class ArrayType {
    UNKNOWN,
    LOGICAL,
    CHAR,
    DOUBLE,
    SINGLE,
    INT8,
    UINT8,
    INT16,
    UINT16,
    INT32,
    UINT32,
    INT64,
    UINT64,
    COMPLEX_DOUBLE,
    COMPLEX_SINGLE,
    COMPLEX_INT8,
    COMPLEX_UINT8,
    COMPLEX_INT16,
    COMPLEX_UINT16,
    COMPLEX_INT32,
    COMPLEX_UINT32,
```

```
    COMPLEX_INT64,
    COMPLEX_UINT64,
    CELL,
    STRUCT,
    OBJECT,
    VALUE_OBJECT,
    HANDLE_OBJECT_REF,
    ENUM,
    SPARSE_LOGICAL,
    SPARSE_DOUBLE,
    SPARSE_COMPLEX_DOUBLE,
    MATLAB_STRING
};
```

C++ Data Type Conversion

MATLAB ArrayType Value	C++ Type	Description
DOUBLE	double	double-precision (64-bit), floating-point number
SINGLE	float	single-precision (32-bit), floating-point number
INT8	int8_t	signed 8-bit integer
UINT8	uint8_t	unsigned 8-bit integer
INT16	int16_t	signed 16-bit integer
UINT16	uint16_t	unsigned 16-bit integer
INT32	int32_t	signed 32-bit integer
UINT32	uint32_t	unsigned 32-bit integer
INT64	int64_t	signed 64-bit integer
UINT64	uint64_t	unsigned 64-bit integer
CHAR	char16_t	16-bit character
LOGICAL	bool	logical
COMPLEX_DOUBLE	std::complex<double>	complex, double-precision (64-bit), floating-point number

MATLAB ArrayType Value	C++ Type	Description
COMPLEX_SINGLE	std::complex<float>	complex, single precision (32-bit), floating-point number
COMPLEX_INT8	std::complex<int8_t>	complex, signed 8-bit integer
COMPLEX_UINT8	std::complex<uint8_t>	complex, unsigned 8-bit integer
COMPLEX_INT16	std::complex<int16_t>	complex, signed 16-bit integer
COMPLEX_UINT16	std::complex<uint16_t>	complex, unsigned 16-bit integer
COMPLEX_INT32	std::complex<int32_t>	complex, signed 32-bit integer
COMPLEX_UINT32	std::complex<uint32_t>	complex, unsigned 32-bit integer
COMPLEX_INT64	std::complex<int64_t>	complex, signed 64-bit integer
COMPLEX_UINT64	std::complex<uint64_t>	complex, unsigned 64-bit integer
CELL	matlab::data::Array	Array containing other Arrays
STRUCT	matlab::data::Struct	Array with named fields that can contain data of varying types and sizes
OBJECT	matlab::data::Object	MATLAB object
VALUE_OBJECT	matlab::data::Object	MATLAB value object
HANDLE_OBJECT_REF	matlab::data::Object	Reference to an existing handle object in MATLAB
ENUM	matlab::data::Enumeration	Array of enumeration values
SPARSE_LOGICAL	bool	Sparse array of logical
SPARSE_DOUBLE	double	Sparse array of double

MATLAB ArrayType Value	C++ Type	Description
SPARSE_COMPLEX_DOUBLE	std::complex<double>	Sparse array of complex double
MATLAB_STRING	matlab::data::MATLABString	MATLAB string

See Also

`ArrayVisitors`

Introduced in R2017b

matlab::data::ArrayVisitors

Functions for defining visitor pattern

Details

Namespace:	`matlab::data`
Include:	`ArrayVisitors.hpp`

Functions

- "apply_visitor" on page 1-31
- "apply_visitor_ref" on page 1-31

apply_visitor

```
auto apply_visitor(Array a, V visitor)
```

TBD

TBD

`InvalidArrayTypeException`

apply_visitor_ref

```
auto apply_visitor_ref(const ArrayRef& a, V visitor)
```

Apply functor V to Array a.

Result of V.

`InvalidArrayTypeException`

See Also

Introduced in R2017b

matlab::data::CellArray

C++ class to access MATLAB cell arrays

Description

A `CellArray` is a `TypedArray` with `Array` as the element type. Use `CellArray` objects to access MATLAB cell arrays. To create a `CellArray`, call `createCellArray`. `CellArray` is defined as:

```
using CellArray = TypedArray<Array>;
```

Class Details

Namespace:	`matlab::data`
Include:	`TypedArray.hpp`

Examples

Create Cell Array

Create a cell array with two elements.

```
#include "MatlabDataArray.hpp"

int main()
{
    using namespace matlab::data;
    ArrayFactory factory;
    CellArray cellArray1 = factory.createCellArray({ 1,2 },
        factory.createCharArray("MATLAB Cell Array"),
        factory.createArray<double>({ 2,2 }, { 1.2, 2.2, 3.2, 4.2 }));
```

```
    return 0;
}
```

See Also

createCellArray

Introduced in R2017b

matlab::data::CharArray

C++ class to access MATLAB character arrays

Description

Use `CharArray` objects to work with MATLAB character arrays. To create a `CharArray`, call `createCharArray`.

Class Details

Namespace:	matlab::data
Base class:	TypedArray<char16_t>
Include:	CharArray.hpp

Constructors

- “Copy Constructors” on page 1-35
- “Copy Assignment Operators” on page 1-36
- “Move Constructors” on page 1-37
- “Move Assignment Operators” on page 1-37

Copy Constructors

```
CharArray(const CharArray& rhs)
```

```
CharArray(const Array& rhs)
```

Creates a shared data copy of a `CharArray` object.

const CharArray& rhs	Value to copy.
const Array& rhs	Value specified as `ArrayType::CHAR` object.

InvalidArrayTypeException	Type of input `Array` is not `ArrayType::CHAR`.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
    CharArray B(A);
    return 0;
}
```

createCharArray on page 1-16

Copy Assignment Operators

`CharArray& operator=(const CharArray& rhs)`

`CharArray& operator=(const Array& rhs)`

Assigns a shared data copy to a `CharArray` object.

const CharArray& rhs	Value to copy.
const Array& rhs	Value specified as `ArrayType::CHAR` object.

CharArray&	Updated instance.

InvalidArrayTypeException	Type of input `Array` is not `ArrayType::CHAR`.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
    CharArray C = factory.createCharArray("");
```

```
    // Arrays A and C refer to the same data.
    C = A;

    return 0;
}
```

Move Constructors

```
CharArray(CharArray&& rhs)

CharArray(Array&& rhs)
```

Moves contents of a `CharArray` object to a new instance.

`CharArray&& rhs`	Value to move.
`Array&& rhs`	Value specified as `ArrayType::CHAR` object.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::CHAR`.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");

    // Move constructor - Creates B, copies data from A. A not valid.
    CharArray B(std::move(A));

    return 0;
}
```

Move Assignment Operators

```
CharArray& operator=(CharArray&& rhs)

CharArray& operator=(Array&& rhs)
```

Assigns the input to this `CharArray` object.

`CharArray&& rhs`	Value to move.
`Array&& rhs`	Value specified as `ArrayType::CHAR` object.

`CharArray&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::CHAR`.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");

    // Move assignment - Data from A moved to C. A no longer valid.
    CharArray C = factory.createCharArray("");
    C = std::move(A);

    return 0;
}
```

Member Functions

- "toUTF16" on page 1-38
- "toAscii" on page 1-39

toUTF16

```
String toUTF16() const
```

`matlab::data::String`	Contents of `CharArray` as `matlab::data::String`.

None

toAscii

`std::string toAscii() const`

`std::string`	Contents of `CharArray` as ASCII string.

`NonAsciiCharInRequestedAsciiOutputException`	Data contains non-ASCII characters.

```
#include "MatlabDataArray.hpp"

int main()
{
    using namespace matlab::data;
    ArrayFactory f;
    auto arr = f.createCharArray("helloworld");
    std::string s = arr.toAscii();

    return 0;
}
```

"Evaluate Mathematical Function in MATLAB"

See Also

"createCharArray" on page 1-16 | `TypedArray` | `matlab::data::String`

Introduced in R2017b

matlab::data::Reference<CharArray>

C++ class to get reference to `CharArray`

Description

The `CharArrayExt` class extends the APIs available to a reference to a `CharArray`.

Class Details

Namespace:	`matlab::data`
Base class:	`Reference<Array>`
Include:	`TypedArrayRef.hpp`

Member Functions

- "toUTF16" on page 1-40
- "toAscii" on page 1-40

toUTF16

```
String toUTF16() const
```

`matlab::data::String`	Contents of reference to `CharArray` as a `utf16` string.

None

toAscii

```
std::string toAscii() const
```

`std::string`	Contents of reference to `CharArray` as `matlab::data::String`.

`NonAsciiCharInRequestedAsciiOutputException`	Data contains non-ASCII characters.

See Also

`CharArray` | `Reference<TypedArray<T>>`

Introduced in R2017b

matlab::data::EnumArray

C++ class to access MATLAB enumeration arrays

Description

Use `EnumArray` objects to access enumeration arrays. To create an `EnumArray`, call `createEnumArray`.

Class Details

Namespace:	`matlab::data`
Base class:	`TypedArray<Enumeration>`
Include:	`EnumArray.hpp`

Constructors

- “Copy Constructors” on page 1-42
- “Copy Assignment Operators” on page 1-43
- “Move Constructors” on page 1-43
- “Move Assignment Operators” on page 1-43

Copy Constructors

```
EnumArray(const EnumArray& rhs)
```

```
EnumArray(const Array& rhs)
```

Creates a shared data copy of an `EnumArray` object.

`const EnumArray& rhs`	Value to copy.
`const Array& rhs`	Value specified as `EnumArray` object.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::ENUM`.

Copy Assignment Operators

`EnumArray& operator=(const EnumArray& rhs)`

`EnumArray& operator=(const Array& rhs)`

Assigns a shared data copy to an `EnumArray` object.

`const EnumArray& rhs`	Value to copy.
`const Array& rhs`	Value specified as `ArrayType::ENUM` object.

`EnumArray&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::ENUM`.

Move Constructors

`EnumArray(EnumArray&& rhs)`

`EnumArray(Array&& rhs)`

Moves contents of an `EnumArray` object to a new instance.

`EnumArray&& rhs`	Value to move.
`Array&& rhs`	Value specified as `ArrayType::ENUM` object.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::ENUM`.

Move Assignment Operators

`EnumArray& operator=(EnumArray&& rhs)`

```
EnumArray& operator=(Array&& rhs)
```

Assigns the input to this `EnumArray` object.

`EnumArray&& rhs`	Value to move.
`Array&& rhs`	Value specified as `ArrayType::ENUM` object.

`EnumArray&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::ENUM`.

Member Functions

getClassName

```
std::string getClassName() const
```

Return class name for this `EnumArray`.

`std::string`	The class name.

None

See Also

"createEnumArray" on page 1-18 | `TypedArray`

Introduced in R2017b

matlab::data::Reference<EnumArray>

C++ class to get reference to `EnumArray`

Description

The `EnumArrayExt` class extends the APIs available to a reference to an `EnumArray`.

Class Details

Namespace:	`matlab::data`
Base class:	`Reference<Array>`
Include:	`TypedArrayRef.hpp`

Member Functions

getClassName

```
std::string getClassName() const
```

Return class name for this reference to an `EnumArray` object.

`std::string`	Class name.

None

See Also

`EnumArray` | `Reference<TypedArray<T>>`

Introduced in R2017b

matlab::data::Enumeration

Element type for MATLAB enumeration arrays

Description

`Enumeration` is the element type for an `EnumArray` object.

Class Details

Namespace:	`matlab::data`
Include:	`Enumeration.hpp`

See Also

`EnumArray`

Topics

"MATLAB Data API Types"

Introduced in R2017b

matlab::Exception

C++ base class for exceptions

Description

All MATLAB C++ exceptions can be caught as `matlab::Exception`.

Class Details

Namespace:	`matlab`
Include:	`Exception.hpp`

See Also

Topics

"MATLAB Data API Exceptions"

Introduced in R2017b

matlab::data::ForwardIterator<T>

Templated C++ class to provide forward iterator support for `StructArray` field names

Description

Use `ForwardIterator` objects to access a range of field name elements in a `StructArray`.

Class Details

Namespace:	`matlab::data`
Include:	`ForwardIterator.hpp`

Template Parameters

`T`	`matlab::data::MATLABFieldIdentifier`

Constructors

- "Copy Constructors" on page 1-48
- "Copy Assignment Operators" on page 1-49

Copy Constructors

```
ForwardIterator(const ForwardIterator<T>& rhs)
```

Creates a shared data copy of a `ForwardIterator<T>` object.

`const ForwardIterator<T>& rhs`	Object to copy.

`ForwardIterator`	New instance.

None

Copy Assignment Operators

```
ForwardIterator<T>& operator=(const ForwardIterator<T>& rhs)
```

Assigns a shared data copy to a ForwardIterator<T> object.

const ForwardIterator<T>& rhs	Object to assign.

ForwardIterator<T>	Updated instance.

None

Other Operators

- "operator++" on page 1-49
- "operator--" on page 1-50
- "operator=" on page 1-50
- "operator!=" on page 1-50
- "operator*" on page 1-51
- "operator->" on page 1-51
- "operator[]" on page 1-51

operator++

```
ForwardIterator<T>& operator++()
```

Pre-increment operator.

ForwardIterator<T>&	Reference to updated value.

None

operator--

`ForwardIterator<T> operator--(int)`

Post-increment operator.

`ForwardIterator<T>`	New object.

None

operator=

`bool operator==(const ForwardIterator<T>& rhs) const`

`const ForwardIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if the iterators point to the same element. Otherwise, returns `false`.

None

operator!=

`bool operator!=(const ForwardIterator<T>& rhs) const`

`const ForwardIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if this iterator points to a different element. Otherwise, returns `false`.

None

operator*

```
reference operator*() const
```

`reference`	Shared copy of element that iterator points to, specified as: • `T&` for arithmetic types. • `Reference<T>` for non-arithmetic types.

None

operator->

```
pointer operator->()
```

`pointer`	Pointer to element pointed to by this iterator, specified as: • `T*` for arithmetic types. • `Reference<T>*` for non-arithmetic types.

None

operator[]

```
reference operator[](const size_t& rhs) const
```

Get a reference using a linear index.

`reference`	Element pointed to by this iterator, specified as: • `T&` for arithmetic types. • `Reference<T>` for non-arithmetic types.

None

See Also

`MATLABFieldIdentifier` | `StructArray`

Introduced in R2017b

matlab::data::GetArrayType

C++ struct TBD

Description

Use `GetArrayType` objects to TBD

Class Details

Namespace:	`matlab::data`
Include:	`GetArrayType.hpp`

See Also

Introduced in R2017b

matlab::data::GetReturnType

C++ struct TBD

Description

Use `GetReturnType` to convert a template argument to the appropriate template argument when creating a `TypedArray`.

Class Details

Namespace:	`matlab::data`
Include:	`GetReturnType.hpp`

See Also

Introduced in R2017b

matlab::data::MATLABFieldIdentifier

C++ class used to identify field names in MATLAB struct array

Description

Class Details

Namespace:	`matlab::data`
Include:	`MATLABFieldIdentifier.hpp`

Constructors

- "Default Constructor" on page 1-55
- "Constructor" on page 1-55
- "Destructor" on page 1-56
- "Copy Constructors" on page 1-56
- "Copy Assignment Operators" on page 1-56
- "Move Constructors" on page 1-57
- "Move Assignment Operators" on page 1-57

Default Constructor

```
MATLABFieldIdentifier()
```

Construct an empty `MATLABFieldIdentifier`.

None

Constructor

```
MATLABFieldIdentifier(std::string str)
```

Construct a `MATLABFieldIdentifier` from `std::string`.

`std::string str`	String that contains the field name.

Destructor

```
~MATLABFieldIdentifier()
```

Destroy a `MATLABFieldIdentifier`.

None

Copy Constructors

```
MATLABFieldIdentifier(const MATLABFieldIdentifier& rhs)
```

Creates a shared data copy of a `MATLABFieldIdentifier` object.

`const MATLABFieldIdentifier& rhs`	Value to copy.

None

Copy Assignment Operators

```
MATLABFieldIdentifier& operator=(MATLABFieldIdentifier const& rhs)
```

Assigns a shared data copy to a `MATLABFieldIdentifier` object.

`MATLABFieldIdentifier const& rhs`	Value to move.

`MATLABFieldIdentifier&`	Updated instance.

None

Move Constructors

`MATLABFieldIdentifier(MATLABFieldIdentifier&& rhs)`

Moves contents a `MATLABFieldIdentifier` object to a new instance.

`MATLABFieldIdentifier&& rhs`	Value to move.

None

Move Assignment Operators

`MATLABFieldIdentifier& operator=(MATLABFieldIdentifier&& rhs)`

`MATLABFieldIdentifier&& rhs`	Value to move.

`MATLABFieldIdentifier&`	Updated instance.

None

Destructor

`~MATLABFieldIdentifier()`

Description

Destroy a `MATLABFieldIdentifier`.

Other Operators

`operator std::string`

```
operator std::string() const
```

`std::string`	Representation of the `MATLABFieldIdentifier` object.

None

Free Functions

`operator==`

```
bool operator==(const MATLABFieldIdentifier& rhs) const
```

Check if two `MATLABFieldIdentifier` objects are identical.

`const MATLABFieldIdentifier& rhs`	Value to be compared.

`bool`	Returns true if the objects are identical. Otherwise, returns false.

None

See Also

`ForwardIterator` | `StructArray`

Introduced in R2017b

matlab::data::MATLABString

Element type for MATLAB string arrays

Description

`MATLABString` is defined as:

```
using MATLABString = Optional<String>;
```

Class Details

Namespace:	`matlab::data`
Include:	`String.hpp`

See Also

`matlab::data::String`

Introduced in R2017b

matlab::data::Reference<MATLABString>

C++ class to get reference to element of `StringArray`

Description

A `Reference<MATLABString>` object is created when using `operator[]` into an `StringArray` or dereferenceing a String arrary iterator.

Class Details

Namespace:	`matlab::data`
Include:	`MATLABStringReferenceExt.hpp`

Cast

String()

```
operator String() const
```

`matlab::data::String`	Element of a `MATLABString` array converted to `String`.

`NotEnoughIndicesProvidedException`	Not enough indices provided.
`InvalidArrayIndexException`	Index provided is not valid for this Array or one of the indices is out of range.
`TooManyIndicesProvidedException`	Too many indices provided.
`std::runtime_error`	Array element does not have a value.

Member Functions

- "bool" on page 1-61
- "has_value" on page 1-61

bool

```
operator bool() const
```

Check whether string contains a value.

`operator`	True, if string contains a value.

`NotEnoughIndicesProvidedException`	Not enough indices provided.
`InvalidArrayIndexException`	Index provided is not valid for this Array or one of the indices is out of range.
`TooManyIndicesProvidedException`	Too many indices provided.

has_value

```
bool has_value() const
```

Check whether string contains a value.

`bool`	True, if string contains a value.

`NotEnoughIndicesProvidedException`	Not enough indices provided.
`InvalidArrayIndexException`	Index provided is not valid for this Array or one of the indices is out of range.
`TooManyIndicesProvidedException`	Too many indices provided.

See Also

Introduced in R2017b

matlab::data::Array

C++ base class for all array types

Description

Use `Array` objects to represent single and multi-dimensional arrays. The `Array` class provides methods for users to be able to query generic information about all arrays, such as dimensions and type, and has methods to create deep copies and shared data copies. Use `ArrayFactory` methods to construct Arrays. Once an `Array` has been constructed, it can be moved or cloned (deep-copied), or a shared-data copy can be created. `Arrays` support copy-on-write semantics.

Class Details

Namespace:	`matlab::data`
Include:	`MDArray.hpp`

Constructors

- “Default Constructor” on page 1-63
- “Copy Constructors” on page 1-64
- “Copy Assignment Operators” on page 1-64
- “Move Constructors” on page 1-64
- “Move Assignment Operators” on page 1-65

Default Constructor

```
Array()
```

None

Copy Constructors

```
Array(const Array& rhs)
```

Creates a shared data copy of an `Array` object.

`const Array& rhs`	Value to copy.

None

Copy Assignment Operators

```
Array& operator=(const Array& rhs)
```

Assigns a shared data copy to an `Array` object.

`const Array& rhsrhs`	Value to copy.

`Array&`	Updated instance.

None

Move Constructors

```
Array(Array&& rhs)
```

Moves contents of an `Array` object to a new instance.

`Array&& rhs`	Value to move.

None

Move Assignment Operators

```
Array& operator=(Array&& rhs)
```

Assigns the input to this `Array` object.

`Array&& rhs`	Value to move.

`Array&`	Updated instance.

None

Destructor

```
virtual ~Array()
```

Indexing Operators

`operator[]`

```
ArrayElementRef<false> operator[](size_t idx)
```

```
ArrayElementRef<true> operator[](size_t idx) const
```

Enables `[]` indexing on `const` and non-`const` arrays. Indexing is 0-based.

`size_t idx`	First array index

`ArrayElementRef<false>`	Temporary object containing the index specified. The return value allows the element of the array to be modified or retrieved.

`ArrayElementRef<true>`	Temporary object containing the index specified. The return value allows the element of the array to be retrieved, but not modified.

None

Member Functions

- "getType" on page 1-66
- "getDimensions" on page 1-66
- "getNumberOfElements" on page 1-66
- "isEmpty" on page 1-67

getType

```
ArrayType getType() const
```

`ArrayType`	Array type.

None

getDimensions

```
ArrayDimensions getDimensions() const
```

`ArrayDimensions`	Vector of each dimension in the array.

None

getNumberOfElements

```
size_t getNumberOfElements() const
```

`size_t`	The number of elements in the array.

None

isEmpty

```
bool isEmpty() const
```

bool	True if array is empty. False if array is not empty.

None

Free Functions

- "getReadOnlyElements" on page 1-67
- "getWritableElements" on page 1-67

getReadOnlyElements

```
template <typename T>
Range<TypedIterator, T const> getReadOnlyElements(const Array& arr)
```

Get a range containing the elements of the `Array`. Iterators contained in the range are `const`.

const Array& arr	Array

Range<TypedIterator, T const>	Range containing `begin` and `end` iterators for the input `Array`.

InvalidArrayTypeException	`Array` does not contain type `T`.

getWritableElements

```
template <typename T>
Range<TypedIterator, T> getWritableElements(Array& arr)
```

Get a range containing the elements of the `Array`. Iterators contained in the range are non-`const`.

`Array& arr`	`Array`

`Range<TypedIterator, T>`	Range containing `begin` and `end` iterators for the input `Array`.

`InvalidArrayTypeException`	`Array` does not contain type `T`.

See Also

`ArrayFactory`

Introduced in R2017b

matlab::data::Object

Element type for MATLAB object arrays

Description

`Object` is the element type for an `ObjectArray`.

Class Details

Namespace:	`matlab::data`
Include:	`Object.hpp`

See Also

`ObjectArray`

Introduced in R2017b

matlab::data::ObjectArray

C++ class to access MATLAB object arrays

Description

Use `ObjectArray` objects to access MATLAB object arrays. You do not create an `ObjectArray`; only MATLAB functions create `ObjectArrays`. You can pass an `ObjectArray` to a MATLAB function.

`ObjectArray` is defined as:

```
using ObjectArray = TypedArray<Object>;
```

Class Details

Namespace:	`matlab::data`
Include:	`ObjectArray.hpp`

See Also

`Object`

Introduced in R2017b

matlab::data::Optional<T>

Templated C++ class representing optional values

Description

Use `Optional` objects to represent values that might or might not exist.

Class Details

Namespace:	`matlab::data`
Include:	`Optional.hpp`

Template Parameters

`T`	Array type, specified as `matlab::data::String`.

Constructors

- “Default Constructors” on page 1-71
- “Copy Constructors” on page 1-71
- “Copy Assignment Operators” on page 1-72
- “Move Constructors” on page 1-72
- “Move Assignment Operators” on page 1-72

Default Constructors

```
optional()
```

Copy Constructors

```
optional(const optional& other)
```

Creates a shared data copy.

`const optional& other`	Value to copy.

None

Copy Assignment Operators

```
optional<T>& operator=(const optional<T>& other)
```

Assigns a shared data copy.

`const optional<T>& other`	Value to copy.

`optional<T>&`	Updated instance.

None

Move Constructors

```
optional(optional&& other)
```

```
optional(T&& value)
```

Moves contents of an `Optional` object to a new instance.

`optional&& other`	Value to move.
`T&& value`	Value of type `T` to move.

None

Move Assignment Operators

```
optional<T>& operator=(optional<T>&& other)
```

```
optional<T>& operator=(T&& value)
```

Assigns the input to this instance.

`optional<T>&& other` `T&& value`	Value to move.

`optional<T>&`	Updated instance.

None

Other Operators

- “operator=” on page 1-73
- “operator->” on page 1-74
- “operator*” on page 1-74
- “operator T” on page 1-74

operator=

```
optional<T>& operator=(nullopt_t)

optional<T>& operator=(const optional<T>& other)

optional<T>& operator=(optional<T>&& other)

optional<T>& operator=(T&& value)

optional<T>& operator=(const T& value)
```

Assignment operators.

`optional<T>&`	Updated instance.

None

operator->

`const T* operator->() const`

`T* operator->()`

`const T*`
 	
`T*`
	Pointer to the element.

`std::runtime_error`	`Optional` object does not contain a value.

operator*

`const T& operator*() const`

`T& operator*()`

`const T&`
 	
`T&`
	Reference to the element.

`std::runtime_error`	`Optional` object does not contain a value.

operator T

`operator T() const`

Cast `optional<T>` value to `T`.

`operator`	Value contained in `optional<T>`, if it exists.

`std::runtime_error`	There is no value.

Member Functions

- “bool” on page 1-75
- “has_value” on page 1-75
- “swap” on page 1-75
- “reset” on page 1-76

bool

```
explicit operator bool() const
```

Check whether object contains a value.

`operator`	True, if object contains a value.

None

has_value

```
bool has_value() const
```

Check whether object contains a value.

`bool`	True, if object contains a value.

None

swap

```
void swap(optional &other)
```

Swap value of this optional instance with value contained in the parameter.

`optional &other`	Value to swap.

None

reset

```
void reset()
```

Reset optional value to missing

None

See Also

Introduced in R2017b

matlab::data::Range<ItType,ElemType>

Templated C++ class to provide range-based operation support

Description

`Range` objects wrap `begin` and `end` functions to enable range-based operations.

Class Details

Namespace:	`matlab::data`
Include:	`Range.hpp`

Template Parameters

`IteratorType`	Iterator type
`ElementType`	Element type

Constructors

- “Constructor” on page 1-77
- “Move Constructors” on page 1-78
- “Move Assignment Operators” on page 1-78

Constructor

```
Range(IteratorType<ElementType> begin, IteratorType<ElementType>
end)
```

Creates a `Range` object.

`IteratorType<ElementType> begin`
 	
`IteratorType<ElementType> end`
	First and last elements of range.

`Range`	New instance.

None

Move Constructors

`Range(Range&& rhs)`

Moves contents of a `Range` object to a new instance.

`Range&& rhs`	Range to move.

`Range`	New instance.

None

Move Assignment Operators

`Range& operator=(Range&& rhs)`

Assigns the input to this `Range` object.

`Range&& rhs`	Range to move.

`Range&`	Updated instance.

None

begin

```
IteratorType<ElementType>& begin()
```

Returns

`IteratorType<ElementType>&`	First element in range.

None

end

```
IteratorType<ElementType>& end()
```

Returns

`IteratorType<ElementType>&`	End of range.

None

See Also

Introduced in R2017b

matlab::data::Reference<T>

Templated C++ class to get references to `Array` elements

Description

A `Reference` object is a reference to an element of an `Array` without making a copy. A `Reference` is:

- Not a shared copy
- Valid as long as the array that contains the reference is valid
- Not thread-safe

Class Details

Namespace:	`matlab::data`
Include:	`Reference.hpp`

Template Parameters

`T`	Type of element referred to, specified as: • `Array` • `Struct` • `Enumeration` • `MATLABString` • All `std::complex` types

Constructors

- "Copy Constructor" on page 1-81
- "Copy Assignment Operators" on page 1-81
- "Move Assignment Operators" on page 1-81

- "Move Constructors" on page 1-81

Copy Constructor

`Reference(const Reference<T>& rhs)`

`const Reference<T>& rhs`	Value to copy.

Copy Assignment Operators

`Reference<T>& operator=(const Reference<T>& rhs)`

`const Reference<T>& rhs`	Value to copy.

`Reference<T>&`	Updated instance.

Move Assignment Operators

`Reference<T>& operator=(Reference<T>&& rhs)`

`Reference<T>&& rhs`	Value to move.

`Reference<T>&`	Updated instance.

None

Move Constructors

`Reference(Reference<T>&& rhs)`

Moves contents of a `Reference` object to a new instance.

`Reference<T>&& rhs`	Value to move.

None

Other Operators

- "operator=" on page 1-82
- "operator<<" on page 1-82
- "operator T()" on page 1-83
- "operator std::string()" on page 1-83

operator=

```
Reference<T>& operator=(T rhs)
```

```
Reference<T>& operator=(std::string rhs)
```

```
Reference<T>& operator=(String rhs)
```

`T rhs`	Value to assign. The array being indexed must be non-`const`.
`std::string rhs`	String to assign. The array must be non-`const` and allow strings to be assigned.
`String rhs`	String to assign to `StringArray`. The array being indexed must be non-`const`.

`Reference<T>&`	Updated instance.

None

operator<<

```
std::ostream& operator <<(std::ostream& os, Reference<T> const& rhs)
```

`std::ostream& os`
`Reference<T> const& rhs`

`std::ostream&`

operator T()

```
operator T() const
```

Cast to element from the array.

`T`	Shared copy of element from the array.

None

operator std::string()

```
operator std::string() const
```

Cast to `std::string` from the array. Makes a copy of the `std::string`. Only valid for types that can be cast to a `std::string`.

`std::string`	The string.

`NonAsciiCharInInputDataException`	Input is `std::string` and contains non-ASCII characters.
`std::runtime_error`	`MATLABString` is missing.

Free Functions

operator==

```
inline bool operator ==(Reference<MATLABString> const& lhs,
std::string const& rhs)
```

```
inline bool operator ==(std::string const& lhs,
Reference<MATLABString> const& rhs)
```

```
inline bool operator ==(Reference<MATLABString> const& lhs, String
const& rhs)
```

```
inline bool operator ==(String const& lhs, Reference<MATLABString>
const& rhs)

inline bool operator ==(Reference<MATLABString> const& lhs,
MATLABString const& rhs)

inline bool operator ==(MATLABString const& lhs,
Reference<MATLABString> const& rhs)

inline bool operator ==(Reference<MATLABString> const& lhs,
Reference<MATLABString> const& rhs)

template<typename T> bool operator ==(Reference<T> const& lhs, T
const& rhs)

template<typename T> bool operator ==(T const& lhs, Reference<T>
const& rhs)

template<typename T> bool operator ==(Reference<T> const& lhs,
Reference<T> const& rhs)
```

`Reference<MATLABString> const& lhs`	`std::string const& rhs`	Values to compare.
`std::string const& lhs`	`Reference<MATLABString> const& rhs`	
`Reference<MATLABString> const& lhs`	`String const& rhs`	
`String const& lhs`	`Reference<MATLABString> const& rhs`	
`Reference<MATLABString> const& lhs`	`MATLABString const& rhs`	
`MATLABString const& lhs`	`Reference<MATLABString> const& rhs`	
`Reference<MATLABString> const& lhs`	`Reference<MATLABString> const& rhs`	
`Reference<T> const& lhs`	`T const& rhs`	
`T const& lhs`	`Reference<T> const& rhs`	
`Reference<T> const& lhs`	`Reference<T> const& rhs`	

`bool`	Returns `true` if values are equal.

`std::runtime_error`	Cannot compare argument to `MATLABString`.

See Also

Topics

"Access C++ Data Array Container Elements"

Introduced in R2017b

matlab::data::SparseArray<T>

Templated C++ class to access data in MATLAB sparse arrays

Description

Use `SparseArray` objects to work with sparse MATLAB arrays. To create a `SparseArray`, call `createSparseArray`.

Class Details

Namespace:	`matlab::data`
Base class:	`matlab::data::Array`
Include:	`SparseArray.hpp`

Template Parameters

`T`	Type of element referred to, specified as: • `bool` • `double` • `std::complex<double>`

Constructors

- "Copy Constructors" on page 1-86
- "Copy Assignment Operators" on page 1-87
- "Move Constructors" on page 1-87
- "Move Assignment Operators" on page 1-88

Copy Constructors

```
SparseArray(const SparseArray<T>& rhs)
```

```
SparseArray(const Array& rhs)
```

Creates a shared data copy of a `SparseArray` object.

`const SparseArray<T>& rhs`	Value to copy.
`const Array& rhs`	Value specified as an `Array` of `ArrayType::SPARSE_LOGICAL`, `ArrayType::SPARSE_DOUBLE`, or `ArrayType::SPARSE_COMPLEX_DOUBLE`.

`InvalidArrayTypeException`	Type of input `Array` is not sparse.

Copy Assignment Operators

```
SparseArray& operator=(const SparseArray<T>& rhs)
```

```
SparseArray& operator=(const Array& rhs)
```

Assigns a shared data copy to a `SparseArray` object.

`const SparseArray<T>& rhs`	Value to copy.
`const Array& rhs`	Value specified as an `Array` of type `ArrayType::SPARSE_LOGICAL`, `ArrayType::SPARSE_DOUBLE`, or `ArrayType::SPARSE_COMPLEX_DOUBLE`.

`SparseArray&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` is not sparse.

Move Constructors

```
SparseArray(SparseArray&& rhs)
```

`SparseArray(Array&& rhs)`

Moves contents a `SparseArray` object to a new instance.

`const SparseArray<T>& rhs`	Value to move.
`const Array& rhs`	Value specified as an `Array` of type `ArrayType::SPARSE_LOGICAL`, `ArrayType::SPARSE_DOUBLE`, or `ArrayType::SPARSE_COMPLEX_DOUBLE`.

`InvalidArrayTypeException`	Type of input `Array` is not sparse.

Move Assignment Operators

`SparseArray& operator=(SparseArray<T>&& rhs)`

`SparseArray& operator=(Array&& rhs)`

Assigns the input to this `SparseArray` object.

`const SparseArray<T>& rhs`	Value to move.
`const Array& rhs`	Value specified as an `Array` of type `ArrayType::SPARSE_LOGICAL`, `ArrayType::SPARSE_DOUBLE`, or `ArrayType::SPARSE_COMPLEX_DOUBLE`.

`SparseArray&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` is not sparse.

Iterators

- "Begin Iterators" on page 1-89
- "End Iterators" on page 1-89

Begin Iterators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

`"iterator"` `"const_iterator"`	Iterator to beginning of array.

None

End Iterators

```
iterator end()
```

```
const_iterator end() const
```

```
const_iterator cend() const
```

`"iterator"` `"const_iterator"`	Iterator to end of array.

None

Member Functions

- "getNumberOfNonZeroElements" on page 1-90

• "getIndex" on page 1-90

getNumberOfNonZeroElements

```
size_t getNumberOfNonZeroElements() const
```

Returns the number of nonzero elements in the array.

`size_t`	Number of nonzero elements in the array.

None

getIndex

```
SparseIndex getIndex(const TypedIterator<T>& it)
```

```
SparseIndex getIndex(const TypedIterator<T const>& it)
```

Returns the row-column coordinates of the nonzero entry that the iterator is pointing to.

`const TypedIterator<T>& it` `const TypedIterator<T const>& it`	Iterator pointing to the current entry in the sparse matrix.

`SparseIndex`	Row-column coordinates of the nonzero entry that the iterator is pointing to. `SparseIndex` is defined as `std::pair<size_t, size_t>`.

None

See Also

`Array` | `createSparseArray`

Introduced in R2017b

matlab::data::Reference<SparseArray<T>>

Templated C++ class to get reference to `SparseArray`

Description

Use the `Reference<SparseArray>` class to get a reference to a `SparseArray` element of a container object, such as a MATLAB structure or cell array.

Class Details

Namespace:	`matlab::data`
Include:	`SparseArrayRef.hpp`

Template Parameters

`T`	Type of elements in `SparseArray`, specified as `bool`, `double`, or `std::complex<double>`.

Iterators

- “Begin Interators” on page 1-92
- “End Interators” on page 1-93

Begin Interators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

`“iterator”`
 	
`“const_iterator”`
	Iterator to beginning of array. Iterates over non-zero elements of the `SparseArray`.

None

End Interators

`iterator end()`

`const_iterator end() const`

`const_iterator cend() const`

`“iterator”`
 	
`“const_iterator”`
	Iterator to end of array.

None

Member Functions

getNumberOfNonZeroElements

`size_t getNumberOfNonZeroElements() const`

Returns the number of nonzero elements in the array. Since sparse arrays only store nonzero elements, this method returns the actual array size. It is different from array dimensions that specify the full array size.

`size_t`	Number of nonzero elements in the array.

None

See Also

Introduced in R2017b

matlab::data::String

Type representing strings as `std::basic_string<char16_t>`

Description

The `String` class defines the element type of a `StringArray`. `String` is defined as:

```
using String = std::basic_string<char16_t>;
```

Class Details

Namespace:	`matlab::data`
Include:	`String.hpp`

See Also

`matlab::data::MATLABString`

Introduced in R2017b

matlab::data::StringArray

C++ class to access MATLAB string arrays

Description

Use `StringArray` objects to access MATLAB string arrays. `StringArray` is defined as:

```
using StringArray = TypedArray<MATLABString>;
```

Class Details

Namespace:	`matlab::data`
Include:	`TypedArray.hpp`

See Also

`matlab::data::MATLABString`

Introduced in R2017b

matlab::data::StructArray

C++ class to access MATLAB struct arrays

Description

Use `StructArray` objects to work with MATLAB struct arrays. To access a field for a single element in the array, use the field name. To create a `StructArray` object, call `createStructArray`.

Class Details

Namespace:	`matlab::data`
Base class:	`TypedArray<Struct>`
Include:	`StructArray.hpp`

Constructors

- "Copy Constructors" on page 1-96
- "Copy Assignment Operators" on page 1-97
- "Move Constructors" on page 1-97
- "Move Assignment Operators" on page 1-98

Copy Constructors

```
StructArray(const StructArray& rhs)
```

```
StructArray(const Array& rhs)
```

Creates a shared data copy of a `StructArray` object.

`const StructArray& rhs`	Value to copy.

`const Array& rhs`	Value specified as `ArrayType::STRUCT` object.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::STRUCT`.

Copy Assignment Operators

```
StructArray& operator=(const StructArray& rhs)
```

```
StructArray& operator=(const Array& rhs)
```

Assigns a shared data copy to a `StructArray` object.

`const StructArray& rhs`	Value to copy.
`const Array& rhs`	Value specified as `ArrayType::STRUCT` object.

`StructArray&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::STRUCT`.

Move Constructors

```
StructArray(StructArray&& rhs)
```

```
StructArray(Array&& rhs)
```

Moves contents of a `StructArray` object to a new instance.

`StructArray&& rhs`	Value to move.
`Array&& rhs`	Value specified as `ArrayType::STRUCT` object.

`InvalidArrayTypeException`	Type of input `Array` is not `ArrayType::STRUCT`.

Move Assignment Operators

```
StructArray& operator=(StructArray&& rhs)
```

Assigns the input to this `StructArray` object.

`StructArray&& rhs`	Value to move.

`StructArray&`	Updated instance.

None

Destructor

```
~StructArray()
```

Description

Free memory for `StructArray` object.

Member Functions

- "getFieldNames" on page 1-98
- "getNumberOfFields" on page 1-99

getFieldNames

```
Range<ForwardIterator, MatlabFieldIdentifier const> getFieldNames() const
```

`Range<ForwardIterator, MatlabFieldIdentifier const>`	Contains `begin` and `end` which enable access to all fields in `StructArray` object.

None

getNumberOfFields

```
size_t getNumberOfFields() const
```

size_t	Number of fields.

None

Examples

Create StructArray

Suppose that you have the following MATLAB structure.

```
s = struct('loc', {'east', 'west'}, 'data', {[1, 2, 3], [4., 5., 6., 7., 8.]})
```

Create a variable containing the data for `loc` `east`.

```
val = s(1).data
```

The following C++ code creates these variables.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;

    StructArray S = factory.createStructArray({ 1,2 }, { "loc", "data" });
    S[0]["loc"] = factory.createCharArray("east");
    S[0]["data"] = factory.createArray<uint8_t>({ 1, 3 }, { 1, 2, 3 });
    S[1]["loc"] = factory.createCharArray("west");
    S[1]["data"] = factory.createArray<double>({ 1, 5 }, { 4., 5., 6., 7., 8. });

    Reference<Array> val = S[0]["data"];
    return 0;
}
```

- “Create Structure Array and Send to MATLAB”

See Also

`MATLABFieldIdentifier` | `Range` | `createStructArray`

Topics

"Create Structure Array and Send to MATLAB"

Introduced in R2017b

matlab::data::Reference<StructArray>

C++ class to get reference to `StructArray`

Description

The `StructArrayExt` class extends the APIs available to a reference to a `StructArray`.

Class Details

Namespace:	`matlab::data`
Base class:	`Reference<Array>`
Include:	`TypedArrayRef.hpp`

Member Functions

- "getFieldNames" on page 1-101
- "getNumberOfFields" on page 1-101

getFieldNames

```
Range<ForwardIterator, MATLABFieldIdentifier const> getFieldNames() const
```

`Range<ForwardIterator, MatlabFieldIdentifier const>`	Contains `begin` and `end` which enables access to all fields in `StructArray` object.

None

getNumberOfFields

```
size_t getNumberOfFields() const
```

`size_t`	Number of fields.

None

See Also

`Reference<TypedArray<T>>` | `StructArray`

Introduced in R2017b

matlab::data::Struct

Element type for MATLAB struct arrays

Description

`Struct` is the element type for a `StructArray` object.

Class Details

Namespace:	`matlab::data`
Include:	`Struct.hpp`

Iterators

- "Begin Iterators" on page 1-103
- "End Iterators" on page 1-103

Begin Iterators

`const_iterator begin() const`

`const_iterator cbegin() const`

`const_iterator`	Iterator to beginning of list of fields, specified as `TypedIterator<Array const>`

None

End Iterators

`const_iterator end() const`

`const_iterator cend() const`

`const_iterator`	Iterator to end of list of fields, specified as `TypedIterator<Array const>`

None

Indexing Operators

operator[]

```
Array operator[](std::string idx) const
```

Enables `[]` indexing on a `StructArray` object. Indexing is 0-based.

`std::string idx`	Field name.

`Array`	Shared copy of `Array` found at specified field.

`InvalidFieldNameException`	Field does not exist in this `StructArray`.

See Also

"createStructArray" on page 1-17 | `StructArray`

Introduced in R2017b

matlab::data::Reference<Struct>

C++ class to get reference to element of `StructArray`

Description

Use the `Reference<Struct>` class to access an element of a `StructArray`.

Class Details

Namespace:	`matlab::data`
Include:	`StructRef.hpp`

Indexing Operators

operator[]

```
Reference<Array> operator[](std::string idx)
```

```
Array operator[](std::string idx) const
```

Index into the Struct with a field name.

`std::string idx`	Field name.

`Reference<Array>`	Reference to `Array` found at specified field.
`Array`	Shared copy of `Array` found at specified field.

`InvalidFieldNameException`	Field does not exist in the struct.

Iterators

- “Begin Iterators” on page 1-106
- “End Iterators” on page 1-106

Begin Iterators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

`iterator`	Iterator to beginning of list of fields, specified as `TypedIterator<T>`.
`const_iterator`	Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`.

None

End Iterators

```
iterator end()
```

```
const_iterator end() const
```

```
const_iterator cend() const
```

`iterator`	Iterator to end of list of fields, specified as `TypedIterator<T>`.
`const_iterator`	Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`.

None

Cast

`Struct()`

```
operator Struct() const
```

Create a shared copy of the `Struct`.

`Struct`	Shared copy.

None

See Also

Introduced in R2017b

matlab::data::TypedArray<T>

Templated C++ class to access array data

Description

The templated `TypedArray` class provides typesafe APIs to handle all array types (except sparse arrays) inside an array. This class defines the following iterator types:

```
using iterator = TypedIterator<T>;
```

```
using const_iterator = TypedIterator<T const>;
```

Class Details

Namespace:	`matlab::data`
Base class:	`matlab::data::Array`
Include:	`TypedArray.hpp`

Template Parameters

`T`	Type of element referred to.

Template Instantiations

double
float
int8_t
uint8_t
int16_t
uint16_t
int32_t
uint32_t
int64_t

uint64_t
char16_t
bool
std::complex<double>
std::complex<float>
std::complex<int8_t>
std::complex<uint8_t>
std::complex<int16_t>
std::complex<uint16_t>
std::complex<int32_t>
std::complex<uint32_t>
std::complex<int64_t>
std::complex<uint64_t>
`matlab::data::Array`
`matlab::data::Struct`
`matlab::data::Enumeration`
`matlab::data::MATLABString`

Constructors

- “Copy Constructor” on page 1-109
- “Copy Assignment Operator” on page 1-110
- “Move Constructor” on page 1-110
- “Move Assignment Operator” on page 1-111

Copy Constructor

```
TypedArray(const TypedArray<T>& rhs)
```

```
TypedArray(const Array& rhs)
```

Creates a shared data copy of the input.

`const TypedArray<T>& rhs`	Value to be copied.
`const Array& rhs`	Value specified as `matlab::data::Array` object.

`InvalidArrayTypeException`	Type of input `Array` does not match the type for `TypedArray<T>`.

Copy Assignment Operator

```
TypedArray<T>& operator=(const TypedArray<T>& rhs)
```

```
TypedArray<T>& operator=(const Array& rhs)
```

Assigns a shared data copy of the input to this `TypedArray<T>`.

`const TypedArray<T>& rhs`	Value to be copied.
`const Array& rhs`	Value specified as `matlab::data::Array` object.

`TypedArray<T>&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` does not match the type for `TypedArray<T>`.

Move Constructor

```
TypedArray(TypedArray<T>&& rhs)
```

```
TypedArray(Array&& rhs)
```

Moves contents of the input to a new instance.

`TypedArray<T>&& rhs`	Value to be moved.
`Array&& rhs`	Value specified as `matlab::data::Array` object.

`InvalidArrayTypeException`	Type of input does not match.

Move Assignment Operator

```
TypedArray<T>& operator=(TypedArray<T>&& rhs)
```

```
TypedArray<T>& operator=(Array&& rhs)
```

Moves the input to this `TypedArray<T>` object.

`TypedArray<T>&& rhs`	Value to move.

`TypedArray<T>&`	Updated instance.

`InvalidArrayTypeException`	Type of input `Array` does not match the type for `TypedArray<T>`.

Destructor

```
virtual ~TypedArray()
```

Iterators

- "Begin Iterators" on page 1-111
- "End Iterators" on page 1-112

Begin Iterators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

`iterator`	Iterator to beginning of array, specified as `TypedIterator<T>`.

`const_iterator`	Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`.

None

End Iterators

`iterator end()`

`const_iterator end() const`

`const_iterator cend() const`

`iterator`	Iterator to end of array, specified as `TypedIterator<T>`.
`const_iterator`	Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`.

None

Indexing Operators

`operator[]`

`ArrayElementTypedRef<T, std::is_const<T>::value> operator[](size_t idx)`

`ArrayElementTypedRef<T, true> operator[](size_t idx) const`

Enables `[]` indexing on a `TypedArray`. Indexing is 0-based.

`size_t idx`	First array index.

`ArrayElementTypedRef<T, std::is_const<T>::value>`	Temporary object containing the index specified. If type `T` is `const`, then the return value allows the element of the array to be retrieved, but not modified. Otherwise, the element can be modified or retrieved.
`ArrayElementTypedRef<T, true>`	Temporary object containing the index specified. The return value allows the element of the array to be retrieved, but not modified.

None

Suppose that you have a cell array `c`. Assign a value to a `Reference<Array>` object and call the member function `getType`.

```
Reference<Array> r = c[0][0];
auto t = c[0][0].getType;
```

Member Functions

release

```
buffer_ptr_t<T> release()
```

Release the underlying buffer from the Array. If the Array is shared, a copy of the buffer is made; otherwise, no copy is made. After the buffer is released, the array contains no elements.

`buffer_ptr_t<T>`	A unique_ptr with the data pointer.

`InvalidArrayTypeException`	This `TypedArray` does not support releasing the buffer.

Examples

Assign Values to Array Elements

Create an array equivalent to the MATLAB array `[1 2; 3 4]`, then replace each element of the array with a single value.

```
#include "MatlabDataArray.hpp"

int main() {
    matlab::data::ArrayFactory factory;
    // Create an array equivalent to the MATLAB array [1 2; 3 4].
    matlab::data::TypedArray<double> D = factory.createArray<double>({ 2,2 }, { 1,3,2,4
    // Change the values.
    for (auto& elem : D) {
        elem = 5.5;
    }
    return 0;
}
```

- "Bring Result of MATLAB Calculation Into C++"

See Also

`Array` | `ArrayType`

Topics

"Bring Result of MATLAB Calculation Into C++"

Introduced in R2017b

matlab::data::Reference<TypedArray<T>>

C++ class to get reference to `TypedArray`

Description

The `Reference<TypedArray<T>>` class extends the APIs available to a reference to an `Array`. It derives from the `Reference<Array>` class and provides iterators and type-safe indexing. `Reference<TypedArray<T>>` is not thread-safe - do not pass references to `TypedArray` objects between threads.

`TypedArrayRef` is defined in `TypedArrayRef.hpp` as:

```
template <typename T>
using TypedArrayRef = Reference<TypedArray<T>>;
```

Class Details

Namespace:	`matlab::data`
Base class:	`Reference<Array>`
Include:	`TypedArrayRef.hpp`

Constructor

```
Reference(const Reference<Array>& rhs)
```

Description

Create a `Reference<TypedArray<T>>` object from a `Reference<Array>` object.

Parameters

`const Reference<Array>& rhs`	Value to copy.

Throws

`TypeMismatchException`	Element of `Array` does not match `<T>`.

Iterators

- "Begin Interators" on page 1-116
- "End Iterators" on page 1-116

Begin Interators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

`iterator`	Iterator to beginning of array, specified as `TypedIterator<T>`.
`const_iterator`	Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`.

None

End Iterators

```
iterator end()
```

```
const_iterator end() const
```

```
const_iterator cend() const
```

`iterator`	Iterator to end of array, specified as `TypedIterator<T>`.
`const_iterator`	Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`.

None

Indexing Operators

operator[]

```
ArrayElementTypedRef<arr_elem_type, std::is_const<T>::value>
operator[](size_t idx)

ArrayElementTypedRef<arr_elem_type, true> operator[](size_t idx)
const
```

Enables `[]` indexing on a reference to an `Array`. Indexing is 0-based.

`size_t idx`	First array index.

`ArrayElementTypedRef<arr_elem_type, std::is_const<T>::value>`	Temporary object containing the index specified. If type `T` is `const`, then the return value allows the element of the array to be retrieved, but not modified. Otherwise, the element can be modified or retrieved.
`ArrayElementTypedRef<arr_elem_type, true>`	Temporary object containing the index specified. The return value allows the element of the array to be retrieved, but not modified.

`InvalidFieldNameException`	Field name is invalid for a struct.

Other Operators

operator=

```
Reference<TypedArray<T>>& operator= (TypedArray<T> rhs)
```

Assign a `TypedArray` to an element of the referenced `Array`. The `Array` being indexed must be non-`const`.

`TypedArray<T> rhs`	Value to assign.

`Reference<TypedArray<T>> &`	Updated instance.

None

See Also

Introduced in R2017b

matlab::data::TypedIterator<T>

Templated C++ class to provide random access iterator

Description

`TypedIterator` is the return type of all `begin` and `end` functions that support random access.

Class Details

Namespace:	`matlab::data`
Include:	`TypedIterator.hpp`

Template Parameters

`T`	Type of element referred to.

Template Instantiations

double
float
int8_t
uint8_t
int16_t
uint16_t
int32_t
uint32_t
int64_t
uint64_t
char16_t
bool
std::complex<double>

```
std::complex<float>
std::complex<int8_t>
std::complex<uint8_t>
std::complex<int16_t>
std::complex<uint16_t>
std::complex<int32_t>
std::complex<uint32_t>
std::complex<int64_t>
std::complex<uint64_t>
matlab::data::Array
matlab::data::Struct
matlab::data::Enumeration
matlab::data::MATLABString
```

Constructors

- “Copy Constructors” on page 1-120
- “Copy Assignment Operators” on page 1-121
- “Move Constructors” on page 1-121
- “Move Assignment Operators” on page 1-121

Copy Constructors

```
TypedIterator(const TypedIterator<T>& rhs)
```

Creates a shared data copy of a `TypedIterator` object.

`const TypedIterator<T>& rhs`	Value to copy.

None

Copy Assignment Operators

```
TypedIterator<T>& operator=(const TypedIterator<T>& rhs)
```

Assigns a shared data copy to a `TypedIterator` object.

`const TypedIterator<T>& rhs`	Value to copy.

`TypedIterator<T>&`	Updated instance.

None

Move Constructors

```
TypedIterator(TypedIterator<T> &&rhs)
```

Moves contents of a `TypedIterator` object to a new instance.

`TypedIterator<T>&& rhs`	Value to move.

None

Move Assignment Operators

```
TypedIterator<T>& operator=(TypedIterator<T>&& rhs)
```

Assigns the input to this `TypedIterator` object.

`TypedIterator<T>&& rhs`	Value to move.

`TypedIterator<T>&`	Updated instance.

None

Other Operators

- “operator++” on page 1-122
- “operator--” on page 1-122
- “operator++” on page 1-123
- “operator--” on page 1-123
- “operator+=” on page 1-123
- “operator-=” on page 1-124
- “operator!=” on page 1-124
- “operator<” on page 1-124
- “operator>” on page 1-125
- “operator<=” on page 1-125
- “operator>=” on page 1-125
- “operator+” on page 1-126
- “operator-” on page 1-126
- “operator-” on page 1-126
- “operator*” on page 1-127
- “operator->” on page 1-127
- “operator[]” on page 1-127

operator++

```
TypedIterator<T>& operator++()
```

Pre-increment operator.

TypedIterator<T>&	Original iterator.

None

operator--

```
TypedIterator<T>& operator--()
```

Pre-decrement operator.

`TypedIterator<T>&`	Original iterator.

None

operator++

```
TypedIterator<T> operator++(int)
```

Post-increment operator.

`TypedIterator<T>`	Copy of original iterator.

None

operator--

```
TypedIterator<T> operator--(int)
```

Post-decrement operator.

`TypedIterator<T>`	Copy of original iterator.

None

operator+=

```
TypedIterator<T>& operator+=(difference_type d)
```

Addition assignment operator.

`difference_type d`	Amount to add, specified as `std::ptrdiff_t`.

`TypedIterator<T>&`	Updated instance.

None

operator-=

```
TypedIterator<T>& operator-=(difference_type d)
```

Subtraction assignment operator.

`difference_type d`	Amount to subtract, specified as `std::ptrdiff_t`.

`TypedIterator<T>&`	Updated instance.

None

operator!=

```
bool operator!=(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if iterators do not point to same element.

None

operator<

```
bool operator<(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if left-side iterator is less than right-side iterator.

operator>

```
bool operator>(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if left-side iterator is greater than right-side iterator.

operator<=

```
bool operator<=(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if left-side iterator is less than or equal to right-side iterator.

None

operator>=

```
bool operator>=(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if left-side iterator is greater than or equal to right-side iterator.

None

operator+

```
TypedIterator<T> operator+(difference_type d) const
```

Creates an iterator that is added to this one by the amount passed in.

`difference_type d`	Amount to add, specified as `std::ptrdiff_t`.

`TypedIterator<T>`	Updated instance.

None

operator-

```
TypedIterator<T> operator-(difference_type d) const
```

Creates an iterator that is decremented from this one by the amount passed in.

`difference_type d`	Amount to subtract, specified as `std::ptrdiff_t`.

`TypedIterator<T>`	Updated instance.

None

operator-

```
difference_type operator-(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`difference_type`	Difference between iterators, specified as `std::ptrdiff_t`.

None

operator*

```
reference operator*() const
```

`reference`	Element pointed to by this iterator, specified as: • `T&` for arithmetic types. • `Reference<T>` for non-arithmetic types.

None

operator->

```
pointer operator->()
```

`pointer`	Pointer to element pointed to by this iterator, specified as: • `T*` for arithmetic types. • `Reference<T>*` for non-arithmetic types.

None

operator[]

```
reference operator[](const size_t& rhs) const
```

Get a reference using a linear index.

`reference`	Element pointed to by this iterator, specified as: • `T&` for arithmetic types. • `Reference<T>` for non-arithmetic types.

None

Free Function

operator==

```
bool operator==(const TypedIterator<T>& rhs) const
```

`const TypedIterator<T>& rhs`	Iterator to compare.

`bool`	Returns `true` if both iterators point to same element.

None

See Also

Introduced in R2017b

matlab::data::apply_visitor

Call Visitor class on arrays

Description

`auto apply_visitor(Array a, V visitor)` dispatch to visitor class operations based on array type.

Include

Namespace:	`matlab::data`
Include	`ArrayVisitors.hpp`

Parameters

`matlab::data::Array a`	The `matlab::data::Array` to operate on with the visitor class.
visitor class V	The user-supplied visitor class.

Return Value

`auto`	Outputs returned by the visitor.

See Also

Topics

"Operate on C++ Arrays Using Visitor Pattern"

Introduced in R2017b

matlab::data::apply_visitor_ref

Call Visitor class on array references

Description

`auto apply_visitor_ref(const ArrayRef& a, V visitor)` dispatch to visitor class operations based on array reference type.

Include

Namespace:	`matlab::data`
Include	`ArrayVisitors.hpp`

Parameters

`const matlab::data::ArrayRef& a`	A `matlab::data::ArrayRef` reference to the array to operate on with the visitor class.
visitor class V	The user-supplied visitor class.

Return Value

`auto`	Outputs returned by the visitor.

See Also

Topics

"Operate on C++ Arrays Using Visitor Pattern"

Introduced in R2017b

matlab::engine::MATLABEngine

Evaluate MATLAB functions from C++ program

Description

The `matlab::engine::MATLABEngine` class uses a MATLAB process as a computational engine for C++. This class provides an interface between the C++ language and MATLAB, enabling you to evaluate MATLAB functions and expressions from C++ programs.

Class Details

Namespace:	`matlab::engine`
Include:	`MatlabEngine.hpp`

Factory Methods

The `matlab::engine::MATLABEngine` class provides methods to start MATLAB and to connect to a shared MATLAB session synchronously or asynchronously.

- `matlab::engine::startMATLAB` — Start MATLAB synchronously
- `matlab::engine::startMATLABAsync` — Start MATLAB asynchronously
- `matlab::engine::connectMATLAB` — Connect to shared MATLAB session synchronously
- `matlab::engine::connectMATLABAsync` — Connect to shared MATLAB session asynchronously

Unsupported Startup Options

The engine does not support these MATLAB startup options:

- `-h`

- `-help`
- `-?`
- `-n`
- `-e`
- `-softwareopengl`
- `-logfile`

For information on MATLAB startup options, see “Commonly Used Startup Options”. For an example of how to use MATLAB startup options when starting engine applications, see “Start MATLAB with Startup Options”.

Method Summary

Member Functions

“feval” on page 1-133	Evaluate MATLAB function with arguments synchronously
“fevalAsync” on page 1-136	Evaluate MATLAB function with arguments asynchronously
“eval” on page 1-138	Evaluate MATLAB statement as a string synchronously
“evalAsync” on page 1-139	Evaluate MATLAB statement as a string asynchronously
“getVariable” on page 1-140	Get variable from the MATLAB base workspace synchronously
“getVariableAsync” on page 1-141	Get variable from the MATLAB base workspace asynchronously
“setVariable” on page 1-142	Put variable into the MATLAB base workspace synchronously
“setVariableAsync” on page 1-142	Put variable into the MATLAB base workspace asynchronously
“getProperty” on page 1-143	Get object property value
“getPropertyAsync” on page 1-144	Get object property value asynchronously

"setProperty" on page 1-145	Set object property value
"setPropertyAsync" on page 1-146	Set object property value asynchronously

Member Function Details

feval

```
std::vector<matlab::data::Array> feval(const matlab::engine::String &function,
   const size_t numReturned,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())

matlab::data::Array feval(const matlab::engine::String &function,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())

matlab::data::Array feval(const matlab::engine::String &function,
   const matlab::data::Array &arg,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())
```

```
ResultType feval(const matlab::engine::String &function,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output,
   const std::shared_ptr<matlab::engine::StreamBuffer> &error,
   RhsArgs&&... rhsArgs )

ResultType feval(const matlab::engine::String &function,
   RhsArgs&&... rhsArgs)
```

Evaluate MATLAB functions with input arguments synchronously. Use feval when you want to pass arguments from C++ to MATLAB and when you want to return a result from MATLAB to C++.

Inputs and outputs can be types defined by the MATLAB Data Array API or can be native C++ types.

`const matlab::engine::String &function`	Name of the MATLAB function or script to evaluate.

const size_t numReturned	Number of returned values
const std::vector<matlab::data::Array> &args	Multiple input arguments to pass to the MATLAB function in a std::vector. The vector is converted to a column array in MATLAB.
const matlab::data::Array arg	Single input argument to pass to the MATLAB function.
const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>()	Stream buffer used to store the standard output from the MATLAB function.
const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>()	Stream buffer used to store the error message from the MATLAB function.
RhsArgs&&... rhsArgs	Native C++ data types used for function inputs. feval accepts scalar inputs of these C++ data types: bool, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, float, double.

std::vector<matlab::data::Array>	Outputs returned from MATLAB function.
matlab::data::Array	Single output returned from MATLAB function.
ResultType	Output returned from MATLAB function as a user-specified type. Can be a std::tuple if returning multiple arguments.

matlab::engine::MATLABNotAvailableException	The MATLAB session is not available.

`matlab::engine::MATLABExecutionException`	There is a MATLAB runtime error in the function.
`matlab::engine::TypeConversionException`	The result of a MATLAB function cannot be converted to the specified type.
`matlab::engine::MATLABSyntaxException`	There is a syntax error in the MATLAB function.

This example passes an array of numeric values to a MATLAB function. The code performs these steps:

- Creates a `matlab::data::Array` with the dimensions 2-by-3 from a vector of numeric values of type double.
- Starts a shared MATLAB session.
- Passes the data array to the MATLAB `sqrt` function and returns the result to C++.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
using namespace matlab::engine;

    std::vector<double> cppData{ 4, 8, 12, 16, 20, 24 };

    // Create a 2-by-3 matlab data array
    matlab::data::ArrayFactory factory;
    auto inputArray = factory.createArray({ 2, 3 }, cppData.cbegin(), cppData.cend());

    // Start MATLAB engine
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();

    // Pass data array to MATLAB sqrt function
    // And return results.
    auto result = matlabPtr->feval(u"sqrt", inputArray);
```

When calling `feval` using native C++ types, the input arguments are restricted to scalar values. For example, this code returns the square root of a scalar value.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

    // Start MATLAB engine synchronously
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();

    // Call sqrt function
    double result = matlabPtr->
        feval<double>(u"sqrt", double(27));
```

For functions that return multiple output arguments, you can use the MATLAB data API or, if using C++ types, a `std::tuple`. For an example, see “Call Function with Native C ++ Types”.

“Call MATLAB Functions from C++”

“MATLAB Data API”

fevalAsync

```
FutureResult<std::vector<matlab::data::Array>> fevalAsync(const matlab::engine::String &function,
   const size_t numReturned,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())

FutureResult<matlab::data::Array> fevalAsync(const matlab::engine::String &function,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())

FutureResult<matlab::data::Array> fevalAsync(const matlab::engine::String &function,
   const matlab::data::Array &arg,
   const std::shared_ptr<matlab::engine::StreamBuffer> & output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> & error = std::shared_ptr<matlab::engine::StreamBuffer>())

FutureResult<ResultType> fevalAsync(const matlab::engine::String &function,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output,
   const std::shared_ptr<matlab::engine::StreamBuffer> &error,
   RhsArgs&&... rhsArgs)

FutureResult<ResultType> fevalAsync(const matlab::engine::String &function,
   RhsArgs&&... rhsArgs)
```

Evaluate MATLAB function with input arguments and returned values asynchronously.

`const matlab::engine::String &function`	Name of the MATLAB function or script to evaluate.
`const size_t numReturned`	Number of returned values
`const std::vector<matlab::data::Array> &args`	Multiple input arguments to pass to the MATLAB function in a `std::vector`. The vector is converted to a column array in MATLAB.

`const matlab::data::Array arg`	Single input argument to pass to the MATLAB function.
`const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>()`	Stream buffer used to store the standard output from the MATLAB function.
`const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>()`	Stream buffer used to store the error message from the MATLAB function.
`RhsArgs&&... rhsArgs`	Native C++ data types used for function inputs. `feval` accepts scalar inputs of these C++ data types: `bool`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `float`, `double`.

`FutureResult`	A `FutureResult` object used to get the result of calling the MATLAB function.

None

This example passes the scalar double 12.7 to the MATLAB `sqrt` function asynchronously. The `FutureResult` is then used to get the result.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
    matlab::data::ArrayFactory factory;
    matlab::data::Array argument = factory.createScalar<double>(12.7);
    FutureResult<matlab::data::Array> future = matlabPtr->
        fevalAsync(convertUTF8StringToUTF16String("sqrt"), std::move(argument));
```

```
    ...
matlab::data::TypedArray<double> result = future.get();
```

"Call Function Asynchronously"

eval

```
void eval(const matlab::engine::String &statement,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer> (),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer> ())
```

Evaluate a MATLAB statement as a string synchronously.

`const matlab::engine::String &statement`	MATLAB statement to evaluate
`const std::shared_ptr<matlab::engine::StreamBuffer> &output`	Stream buffer used to store the standard output from the MATLAB statement.
`const std::shared_ptr<matlab::engine::StreamBuffer> &error`	Stream buffer used to store the error message from the MATLAB command.

`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.
`matlab::engine::MATLABExecutionException`	There is a runtime error in the MATLAB statement.
`matlab::engine::MATLABSyntaxException`	There is a syntax error in the MATLAB statement.

This example evaluates the following MATLAB statement.

```
a = sqrt(12.7);
```

The statement creates the variable `a` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlabPtr->eval(convertUTF8StringToUTF16String("a = sqrt(12.7);"));
```

"Evaluate MATLAB Statements from C++"

evalAsync

```
FutureResult<void> evalAsync(const matlab::engine::String &str,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer> (),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer> ())
```

Evaluate a MATLAB statement as a string asynchronously.

`const String& str`	MATLAB statement to evaluate
`const std::shared_ptr<matlab::engine::StreamBuffer> & output`	Stream buffer used to store the standard output from the MATLAB statement.
`const std::shared_ptr<matlab::engine::StreamBuffer> & error`	Stream buffer used to store the error message from the MATLAB command.

`FutureResult`	A `FutureResult` object used to wait for the completion of the MATLAB statement.

None

This example evaluates the following MATLAB statement asynchronously.

```
a = sqrt(12.7);
```

The statement creates the variable `a` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
FutureResult<void> future = matlabPtr->
   evalAsync(convertUTF8StringToUTF16String("a = sqrt(12.7);"));
```

"Evaluate MATLAB Statements from C++"

getVariable

```
matlab::data::Array getVariable(const matlab::engine::String &varName,
    WorkspaceType workspaceType = WorkspaceType::BASE)
```

Get a variable from the MATLAB base or global workspace.

`const matlab::engine::String& varName`	Name of a variable in the MATLAB workspace
`WorkspaceType workspaceType = WorkspaceType::BASE`	MATLAB workspace (BASE or GLOBAL) to get the variable from. For more information, see `global`.

`matlab::data::Array`	Variable obtained from the MATLAB base or global workspace

`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.
`matlab::engine::MATLABExecutionException`	The requested variable does not exist in the specified MATLAB base or global workspace.

This example gets a variable named `varName` from the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::Array varName = matlabPtr->getVariable(convertUTF8StringToUTF16String("varName"));
```

"Pass Variables from MATLAB to C++"

getVariableAsync

```
FutureResult<matlab::data::Array> getVariableAsync(const matlab::engine::String &varName,
    WorkspaceType workspaceType = WorkspaceType::BASE)
```

Get a variable from the MATLAB base or global workspace asynchronously.

`const matlab::engine::String& varName`	Name of the variable in MATLAB workspace.
`WorkspaceType workspaceType = WorkspaceType::BASE`	MATLAB workspace (BASE or GLOBAL) to get the variable from. For more information, see `global`.

`FutureResult`	A `FutureResult` object that you can use to get the variable obtained from the MATLAB workspace as a `matlab.data.Array`.

None

This example gets a variable named `varName` from the MATLAB base workspace asynchronously.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
FutureResult<matlab::data::Array> future = matlabPtr->
   getVariableAsync(convertUTF8StringToUTF16String("varName"));
...
matlab::data::Array varName = future.get();
```

"Pass Variables from MATLAB to C++"

setVariable

```
void setVariable(const matlab::engine::String &varName,
   const matlab::data::Array &var,
   WorkspaceType workspaceType = WorkspaceType::BASE)
```

Put a variable into the MATLAB base or global workspace. If a variable with the same name exists in the MATLAB workspace, `setVariable` overwrites it.

`const matlab::engine::String& varName`	Name of the variable to create in the MATLAB workspace
`const matlab::data::Array var`	Value of the variable to create in the MATLAB workspace
`WorkspaceType workspaceType = WorkspaceType::BASE`	Put the variable in the MATLAB BASE or GLOBAL workspace. For more information, see `global`.

`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.

This example puts the variable named `data` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::Array data = factory.createArray<double>({ 1, 3 }, { 4, 8, 6 });
matlabPtr->setVariable(convertUTF8StringToUTF16String("data"), data);
```

“Pass Variables from C++ to MATLAB”

setVariableAsync

```
FutureResult<void> setVariableAsync(const matlab::engine::String &varName,
   const matlab::data::Array var,
   WorkspaceType workspaceType = WorkspaceType::BASE)
```

Put a variable into the MATLAB base or global workspace asynchronously. If a variable with the same name exists in the MATLAB base workspace, `setVariableAsync` overwrites it.

`const matlab::engine::String& varName`	Name of the variable to create in the MATLAB workspace
`const matlab::data::Array var`	Value of the variable to create in the MATLAB workspace
`WorkspaceType workspaceType = WorkspaceType::BASE`	Put the variable in the MATLAB BASE or GLOBAL workspace. For more information, see `global`.

None

This example puts the variable named `data` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::Array data = factory.createArray<double>({ 1, 3 }, { 4., 8., 6. });
FutureResult<void> future = matlabPtr->
   setVariableAsync(convertUTF8StringToUTF16String("data"), data);
```

“Pass Variables from MATLAB to C++”

getProperty

```
matlab::data::Array getProperty(const matlab::data::Array &object,
   const String &propertyName)
```

Get the value of an object property

`const matlab::data::Array &object`	MATLAB object
`const String &propertyName`	Name of the property

`matlab::data::Array`	Value of the named property

`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.
`matlab::engine::MATLABExecutionException`	The property does not exist.

This example evaluates a MATLAB statement in a try/catch block using `MATLABEngine::eval`. The `MATLABEngine::getVariable` member function returns the exception object. `MATLABEngine::getProperty` returns the exception `message` property value as a `matlab::data::CharArray`.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;
```

```
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlabPtr->eval(convertUTF8StringToUTF16String("try; surf(4); catch me; end"));
matlab::data::Array mException = matlabPtr->
    getVariable(convertUTF8StringToUTF16String("me"));
matlab::data::CharArray message = matlabPtr->
    getProperty(mException, convertUTF8StringToUTF16String("message"));
std::cout << "messages is: " << message.toAscii() << std::endl;
```

“Get MATLAB Objects and Access Properties”

getPropertyAsync

```
FutureResult<matlab::data::Array> getPropertyAsync(const matlab::data::Array &object,
   const String &propertyName)
```

Get the value of an object property asynchronously

`const matlab::data::Array &object`	MATLAB object
`const String &propertyName`	Name of the property.

`FutureResult`	`FutureResult` object that is used to synchronize the operation.

None

This example evaluates a MATLAB statement in a try/catch block using `MATLABEngine::eval`. The `MATLABEngine::getVariable` member function returns the exception object. `MATLABEngine::getPropertyAsync` returns a `FutureResult` that you use to get the exception `message` property value as a `matlab::data::CharArray`.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlabPtr->eval(convertUTF8StringToUTF16String("try;surf(4);catch me;end"));
matlab::data::Array mException = matlabPtr->
    getVariable(convertUTF8StringToUTF16String("me"));
FutureResult<matlab::data::Array> future = matlabPtr->
    getPropertyAsync(mException, convertUTF8StringToUTF16String("message"));
matlab::data::CharArray message = future.get();
std::cout << "messages is: " << message.toAscii() << std::endl;
```

"Get MATLAB Objects and Access Properties"

setProperty

```
void setProperty(matlab::data::Array &object, const String &propertyName,
   const matlab::data::Array &propertyValue)
```

Set the value of an object property

`matlab::data::Array &object`	MATLAB object

`const String &propertyName`	Name of the property to set
`const matlab::data::Array &propertyValue`	Value assigned to the property

`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.
`matlab::engine::MATLABExecutionException`	The property does not exist.

This example shows how to set a MATLAB object property. It creates a MATLAB graph and returns the line handle object. Setting the value of the line `LineStyle` property to the character `:` changes the property value of the line object in MATLAB and updates the line style of the graph.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::ArrayFactory factory;
matlab::data::Array yData = factory.createArray<double>({ 1, 5 }, { 4.0, 11.0, 4.7, 36.2, 72.3 });
matlab::data::Array lineHandle = matlabPtr->feval(convertUTF8StringToUTF16String("plot"), yData);
matlab::data::CharArray lineStyle = factory.createCharArray(":");
matlabPtr->setProperty(lineHandle, convertUTF8StringToUTF16String("LineStyle"), lineStyle);
```

“Set Property on MATLAB Object”

setPropertyAsync

```
FutureResult<void> setPropertyAsync(matlab::data::Array &object,
 const String &propertyName,
 const matlab::data::Array &propertyValue)
```

Set the value of an object property asynchronously.

`matlab::data::Array &object`	MATLAB object
`const String &propertyName`	Name of the property to set

`const matlab::data::Array &propertyValue`	Value assigned to the property.

None

This example shows how to set a MATLAB object property asynchronously. It creates a MATLAB graph and returns the line handle object. Setting the line `LineStyle` property to the character `:` changes the property value of the object in MATLAB and updates the line style of the graph.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::ArrayFactory factory;
matlab::data::Array yData = factory.createArray<double>({ 1, 5 }, { 4.0, 11.0, 4.7, 36.2, 72.3 });
matlab::data::Array lineHandle = matlabPtr->feval(convertUTF8StringToUTF16String("plot"), yData);
matlab::data::CharArray lineStyle = factory.createCharArray(":");
FutureResult<void> future = matlabPtr->
   setPropertyAsync(lineHandle, convertUTF8StringToUTF16String("LineStyle"), lineStyle);
```

“Set Property on MATLAB Object”

See Also

Introduced in R2017b

matlab::engine::connectMATLAB

Connect to shared MATLAB session synchronously

Description

```
std::unique_ptr<MATLABEngine> connectMATLAB()
```

```
std::unique_ptr<MATLABEngine> connectMATLAB(const
matlab::engine::String& name)
```

Connect synchronously to a shared MATLAB session on the local machine.

- If you specify the name of a shared MATLAB session, but the engine cannot find a session with that name, the engine throws an exception.
- If you do not specify a name and there is no shared MATLAB session available, the engine starts a new shared MATLAB session. The MATLAB desktop is not started.
- If you do not specify a name and there are shared MATLAB sessions available, the engine connects to the first available session.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Parameters

`const matlab::engine::String& name`	Name of the shared MATLAB session

Return Value

`std::unique_ptr<MATLABEngine>`	Pointer to a `MATLABEngine` object

Exceptions

`matlab::engine::EngineException`	Throws exception if function fails to connect to the specified MATLAB session.

Examples

Connect to Shared MATLAB Session

Connect to a shared MATLAB session named `my_matlab`.

```
std::unique_ptr<MATLABEngine> matlabPrt =
   connectMATLAB(convertUTF8StringToUTF16String("my_matlab"));
```

- “Start MATLAB Sessions from C++”

See Also

`matlab::engine::connectMatlabAsync`

Topics

“Start MATLAB Sessions from C++”

Introduced in R2017b

matlab::engine::connectMATLABAsync

Connect to shared MATLAB session asynchronously

Description

```
FutureResult<std::unique_ptr<MATLABEngine>> connectMATLABAsync()
```

```
FutureResult<std::unique_ptr<MATLABEngine>> connectMATLABAsync(const
matlab::engine::String& name)
```

Connect asynchronously to a shared MATLAB session on the local machine.

- If you specify the name of a shared MATLAB session, but the engine cannot find a session with that name, the engine throws an exception.
- If you do not specify a name and there is no shared MATLAB session available, the engine starts a new shared MATLAB session. The MATLAB desktop is not started.
- If you do not specify a name and there are shared MATLAB sessions available, the engine connects to the first available session.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Parameters

`const matlab::engine::String& name`	Name of the shared MATLAB session

Return Value

`FutureResult<std::unique_ptr<MATLABEngine>>`	A `FutureResult` object that you can use to get the pointer to the `MATLABEngine`

Examples

Connect to Shared MATLAB Session Asynchronously

Connect to a shared MATLAB session named `my_matlab` asynchronously. Use the `FutureResult` `get` method to retrieve the pointer to the `MATLABEngine` object.

```
FutureResults<std::unique_ptr<MATLABEngine>> future =
   connectMATLABAsync(convertUTF8StringToUTF16String("my_matlab"));
...
std::unique_ptr<MATLABEngine> matlabPtr = future.get();
```

- "Connect C++ to Running MATLAB Session"

See Also

`matlab::engine::connectMatlab`

Topics

"Connect C++ to Running MATLAB Session"

Introduced in R2017b

matlab::engine::convertUTF8StringToUTF16String

Convert UTF-8 string to UTF-16 string

Description

```
std::basic_string<char16_t> convertUTF8StringToUTF16String(const
std::string& utf8string)
```

Convert a UTF-8 string to a UTF-16 string. Use this function to convert ASCII strings to `matlab::engine::String` strings, which are required by MATLAB C++ Engine functions.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Parameters

`const std::string& utf8string`	A UTF-8 string

Return Value

`std::basic_string<char16_t>`	A UTF-16 string

Exceptions

`matlab::engine::OutofMemoryException`	The function failed to allocate memory.

matlab::engine::TypeConversionException	The input type cannot be converted to std::basic_string<char16_t>.

Examples

Convert String

Convert a UTF-8 string to a matlab::engine::String (UTF-16 string).

```
matlab::engine::String matlabStatement =
   convertUTF8StringToUTF16String("sqrt(12.7);");
```

See Also

matlab::engine::String | matlab::engine::convertUTF16StringToUTF8String

Introduced in R2017b

matlab::engine::convertUTF16StringToUTF8String

Convert UTF-16 string to UTF-8 string

Description

```
std::string convertUTF16StringToUTF8String(const
std::basic_string<char16_t>& utf16string)
```

Convert a UTF-16 string to a UTF-8 string.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Parameters

`const std::basic_string<char16_t>& utf16string`	A UTF-16 string

Return Value

`std::string`	A UTF-8 string

Exceptions

`matlab::engine::OutofMemoryException`	The function failed to allocate memory.

`matlab::engine::TypeConversionException`	The input type cannot be converted to `std::string`.

Examples

Convert String

Convert a `matlab::engine::String` (UTF-16 string) to a `std::string` (UTF-8 string).

```
matlab::engine::String matlabStatement = (u"sqrt(12.7);");
std::string cmdString = convertUTF16StringToUTF8String(matlabStatement);
```

See Also

`matlab::engine::String` | `matlab::engine::convertUTF8StringToUTF16String`

Introduced in R2017b

matlab::engine::findMATLAB

Find shared MATLAB sessions synchronously

Description

`std::vector<String> findMATLAB()`

Find all shared MATLAB sessions on the local machine.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Parameters

None

Return Value

`std::vector<String>`	A vector of the names of all shared MATLAB sessions on the local machine, or an empty vector if no shared MATLAB sessions are available

Exceptions

`matlab::engine::EngineException`	Throws exception if the call fails while searching for shared MATLAB sessions.

Examples

Find Shared MATLAB Session Synchronously

```
std::vector<String> names = findMATLAB();
```

See Also

`matlab::engine::findMATLABAsync`

Introduced in R2017b

matlab::engine::findMATLABAsync

Find shared MATLAB sessions asynchronously

Description

```
FutureResult<std::vector<String>> findMATLABAsync()
```

Find all shared MATLAB sessions on the local machine asynchronously.

Include

Namespace:	matlab::engine
Include	MatlabEngine.hpp

Parameters

None

Return Value

FutureResult<std::vector<String>>	A FutureResult object that you can use to get the names of shared MATLAB sessions on the local machine.

Examples

Find Shared MATLAB Session Asynchronously

Find the names of all shared MATLAB sessions on the local machine asynchronously. Use the `FutureResult` `get` method to retrieve the names.

```
FutureResult<std::vector<String>> futureNames = findMATLABAsync();
...
std::vector<String> matlabSessions = futureNames.get();
```

See Also

`matlab::engine::findMATLAB`

Introduced in R2017b

matlab::engine::FutureResult

Retrieve result from asynchronous operation

Description

A future result is an object that you can use to retrieve the result of MATLAB functions or statements. The `FutureResult` class provides all member functions of the C++ `std::future` class.

Class Details

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Constructor Summary

Create a `FutureResult` object using these asynchronous functions:

- Asynchronous member functions defined by `matlab::engine::MATLABEngine`.
- `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, and `matlab::engine::findMATLABAsync`.

Method Summary

Member Functions

“cancel” on page 1-161	Cancel the operation held by the `FutureResult` object.

Member Functions Delegated to std::future
`operator=`, `share`, `get`, `wait`, `wait_for`, `wait_until`

`matlab::engine::EngineException`	Cannot start or connect to MATLAB session.
`matlab::engine::CancelException`	Execution of command is canceled.
`matlab::engine::InterruptedException`	Evaluation of command is interrupted.
`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.
`matlab::engine::MATLABSyntaxException`	There is a syntax error in the MATLAB function.
`matlab::engine::MATLABExecutionException`	MATLAB runtime error in the function.
`matlab::engine::TypeConversionException`	The result from a MATLAB function cannot be converted to the specified type.

Method Details

cancel

```
bool FutureResult::cancel(bool allowInterrupt = true);
```

Cancel the evaluation of the MATLAB function or statement. You cannot cancel asynchronous operations that use: `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, or `matlab::engine::findMATLABAsync`.

`bool allowInterrupt`	If false, do not interrupt if execution had already begun.

`bool`	Was command canceled if execution had already begun.

```
bool flag = future.cancel();
```

No exceptions thrown

See Also

Topics

"Call Function Asynchronously"

Introduced in R2017b

matlab::engine::startMATLAB

Start MATLAB synchronously

Description

```
std::unique_ptr<MATLABEngine> startMATLAB(const std::vector<String>&
options = std::vector<String>())
```

Start MATLAB synchronously in a separate process with optional MATLAB startup options.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Parameters

`const std::vector<String>& options`	Options used to start MATLAB. See “Specify Startup Options”.

Return Value

`std::unique_ptr<MATLABEngine>`	Pointer to the `MATLABEngine` object

Exceptions

`matlab::engine::EngineException`	MATLAB failed to start.

Examples

Start MATLAB Synchronously

Start MATLAB synchronously and return a unique pointer to the `MATLABEngine` object.

```
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
```

Start MATLAB with Startup Options

Start MATLAB with the `-nojvm` option and return a unique pointer to the `MATLABEngine` object.

```
std::vector<String> optionVec;
optionVec.push_back(convertUTF8StringToUTF16String("-nojvm"));
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB(optionVec);
```

- "Start MATLAB Sessions from C++"

See Also

`matlab::engine::MATLABEngine` | `matlab::engine::startMATLABAsync`

Topics

"Start MATLAB Sessions from C++"

Introduced in R2017b

matlab::engine::startMATLABAsync

Start MATLAB asynchronously

Description

```
FutureResult<std::unique_ptr<MATLABEngine>> startMATLABAsync(const
std::vector<String>& options = std::vector<String>())
```

Start MATLAB asynchronously in a separate process with optional MATLAB startup options.

Include

Namespace:	matlab::engine
Include	MatlabEngine.hpp

Parameters

const std::vector<String>& options	Startup options used to launch MATLAB

Return Value

FutureResult<std::unique_ptr<MATLABEngine>>	A FutureResult object used to get the pointer to the MATLABEngine

Examples

Start MATLAB Asynchronously

Start MATLAB asynchronously and return a `FutureResult` object. Use the `FutureResult` to get a pointer to the `MATLABEngine` object.

```
FutureResult<std::unique_ptr<MATLABEngine>> matlabFuture = startMATLAB();
...
std::unique_ptr<MATLABEngine> matlabPtr = matlabFuture.get();
```

- “Specify Startup Options”

See Also

`matlab::engine::startMATLAB`

Topics

“Specify Startup Options”

Introduced in R2017b

matlab::engine::terminateEngineClient

Free engine resources during runtime

Description

`void matlab::engine::terminateEngineClient` releases all MATLAB engine resources during runtime when you no longer need the MATLAB engine in your application program.

Note Programs cannot start a new MATLAB engine or connect to a shared MATLAB session after calling `terminateEngineClient`.

Include

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Examples

Terminate the engine session to free resources.

```
// Start MATLAB session
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
...
// Terminate MATLAB session
matlab::engine::terminateEngineClient();
```

See Also

`matlab::engine::startMATLAB`

Introduced in R2017b

matlab::engine::WorkspaceType

Type of MATLAB workspace

Description

The `matlab::engine::WorkspaceType` enum class specifies the MATLAB workspace to pass variables to or get variables from.

`BASE`	Variables scoped to the MATLAB base workspace (command line and nonfunction scripts)
`GLOBAL`	Variables scoped to the MATLAB global workspace (command line, functions, and scripts)

MATLAB scopes variables by workspace. Variables that are scoped to the base workspace must be passed to functions as arguments. Variables scoped to the global workspace can be accessed by any function that defines the specific variable name as global.

Class Details

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Examples

This example:

- Connects to a shared MATLAB session
- Creates a `matlab::data::Array` containing numeric values of type double
- Puts the array in the MATLAB global workspace

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

static void putGlobalVar() {
```

```
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(convertUTF8StringToUTF16String("myMatlabEngine"));
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create matlab data array factory
    matlab::data::ArrayFactory factory;

    // Create data variable
    matlab::data::Array data = factory.createArray<double>
        ({ 1, 5 }, { 4.0, 11.0, 4.7, 36.2, 72.3 });

    // Put data variable in MATLAB global workspace
    matlabPtr->setVariable(convertUTF8StringToUTF16String("data"), data, WorkspaceType::GLOBAL);
}
```

See Also

`matlab::data::ArrayFactory` | `matlab::engine::MATLABEngine` | `matlab::engine::convertUTF8StringToUTF16String`

Topics

"Pass Variables from C++ to MATLAB"
"Pass Variables from MATLAB to C++"

Introduced in R2017b

matlab::engine::String

Define UTF16 string

Description

Type definition for `std::basic_string<char16_t>`.

Examples

This example defines a variable containing the name of a shared MATLAB session. Pass this string to the `matlab::engine::connectMATLAB` function to connect to the named session.

```
matlab::engine::String session(convertUTF8StringToUTF16String("myMatlabEngine"));
  std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);
```

See Also

`matlab::engine::convertUTF16StringToUTF8String` | `matlab::engine::convertUTF8StringToUTF16String`

Topics

"MATLAB Engine API for C++"
"Connect C++ to Running MATLAB Session"

Introduced in R2017b

matlab::engine::StreamBuffer

Define stream buffer

Description

Type definition for `std::basic_streafbuf<char16_t>`.

Examples

This example defines string buffers to return output from the evaluation of a MATLAB function by the `MATLABEngine::eval` member function. This function uses a buffer derived from `matlab::engine::StreamBuffer` to return output from MATLAB to C++.

```
#include "MatlabEngine.hpp"
#include "MatlabDataArray.hpp"
#include <iostream>

using namespace matlab::engine;
using SBuf = std::basic_stringbuf<char16_t>;

void printFromBuf(const std::shared_ptr<SBuf> buf)
{
    //Get text from buf
    auto text_ = buf->str();
    std::cout << "*" << convertUTF16StringToUTF8String(text_)
        << "*" << std::endl;
}

int main() {

    //Create Array factory
    matlab::data::ArrayFactory factory;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(convertUTF8StringToUTF16String("myMatlabEngine"));
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    auto outBuf = std::make_shared<SBuf>();
    auto errBuf = std::make_shared<SBuf>();

    matlabPtr->eval(convertUTF8StringToUTF16String("matlab.engine.engineName"), outBuf, errBuf);
    printFromBuf(outBuf);
    printFromBuf(errBuf);
    return 0;
}
```

See Also

`matlab::engine::connectMATLAB` | `matlab::engine::convertUTF16StringToUTF8String` | `matlab::engine::convertUTF8StringToUTF16String`

Topics

"Redirect MATLAB Command Window Output to C++"

Introduced in R2017b

matlab::engine::SharedFutureResult

Retrieve result from asynchronous operation as shared future

Description

A shared future result is an object that you use to retrieve the result of MATLAB functions or statements any number of times.

Class Details

Namespace:	`matlab::engine`
Include	`MatlabEngine.hpp`

Constructor Summary

Create a `FutureResult` object using one of these asynchronous functions:

- Asynchronous member functions defined by `matlab::engine::MATLABEngine`.
- `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, and `matlab::engine::findMATLABAsync`.

Method Summary

Member Functions

"cancel" on page 1-161	Cancel the operation held by the `FutureResult` object.

Member Function Delegated to std::shared_future
`operator=`, `get`, `valid`, `wait`, `wait_for`, `wait_until`

`matlab::engine::EngineException`	Cannot start or connect to MATLAB session.
`matlab::engine::CancelException`	Execution of command is canceled.
`matlab::engine::InterruptedException`	Evaluation of command is interrupted.
`matlab::engine::MATLABNotAvailableException`	The MATLAB session is not available.
`matlab::engine::MATLABSyntaxException`	There is a syntax error in the MATLAB function.
`matlab::engine::MATLABExecutionException`	MATLAB runtime error in the function.
`matlab::engine::TypeConversionException`	The result from a MATLAB function cannot be converted to the specified type.

Method Details

cancel

```
bool FutureResult::cancel(bool allowInterrupt = true);
```

Cancel the evaluation of the MATLAB function or statement.

Note that you cannot cancel asynchronous start, connection, or find operations, which are initiated using these functions: `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, or `matlab::engine::findMATLABAsync`.

`bool allowInterrupt`	If false, do not interrupt if execution has already begun.

`bool`	True if the MATLAB command can be canceled

```
bool flag = future.cancel();
```

None

See Also

`matlab::engine::FutureResult`

Topics

"Call Function Asynchronously"

Introduced in R2017b

com.mathworks.engine.MatlabEngine class

Package: com.mathworks.engine

Java class using MATLAB as a computational engine

Description

The `com.mathworks.engine.MatlabEngine` class uses a MATLAB process as a computational engine for Java®. This class provides an interface between the Java language and MATLAB, enabling you to evaluate MATLAB functions and expressions from Java.

Constructor Summary

The `MatlabEngine` class provides static methods to start MATLAB and to connect to a shared MATLAB session synchronously or asynchronously. Only these static methods can instantiate this class:

- Start MATLAB synchronously — "startMatlab" on page 1-178
- Connect to shared MATLAB session synchronously — "connectMatlab" on page 1-180
- Start MATLAB asynchronously — "startMatlabAsync" on page 1-179
- Connect to shared MATLAB session asynchronously — "connectMatlabAsync" on page 1-181

Unsupported Startup Options

The engine does not support these MATLAB startup options:

- `-h`
- `-help`
- `-?`
- `-n`

- `-e`
- `-softwareopengl`
- `-logfile`

For information on MATLAB startup options, see "Commonly Used Startup Options".

Method Summary

Static Methods

"startMatlab" on page 1-178	Start MATLAB synchronously.
"startMatlabAsync" on page 1-179	Start MATLAB asynchronously.
"findMatlab" on page 1-179	Find all available shared MATLAB sessions from a local machine synchronously.
"findMatlabAsync" on page 1-180	Find all available shared MATLAB sessions from a local machine asynchronously.
"connectMatlab" on page 1-180	Connect to a shared MATLAB session on a local machine synchronously.
"connectMatlabAsync" on page 1-181	Connect to a shared MATLAB session on a local machine asynchronously.

Member Variable

NULL_WRITER	Use a writer that ignores the contents from the MATLAB command window.

Member Functions

"feval" on page 1-182	Evaluate a MATLAB function with arguments synchronously.
"fevalAsync" on page 1-183	Evaluate a MATLAB function with arguments asynchronously.
"eval" on page 1-184	Evaluate a MATLAB expression as a string synchronously.

"evalAsync" on page 1-185	Evaluate a MATLAB expression as a string asynchronously.
"getVariable" on page 1-186	Get a variable from the MATLAB base workspace synchronously.
"getVariableAsync" on page 1-187	Get a variable from the MATLAB base workspace asynchronously.
"putVariable" on page 1-187	Put a variable into the MATLAB base workspace synchronously.
"putVariableAsync" on page 1-188	Put a variable into the MATLAB base workspace asynchronously.
"disconnect" on page 1-188	Disconnect from the current MATLAB session synchronously.
"disconnectAsync" on page 1-189	Disconnect from the current MATLAB session asynchronously.
"quit" on page 1-189	Force the shutdown of the current MATLAB session synchronously.
"quitAsync" on page 1-189	Force the shutdown of the current MATLAB session asynchronously.
"close" on page 1-190	Disconnect or terminate the current MATLAB session.

Method Details

startMatlab

```
static MatlabEngine startMatlab(String[] options)
```

```
static MatlabEngine startMatlab()
```

Start MATLAB synchronously.

`String[] options`	Startup options used to start MATLAB engine. For options, see "Startup and Shutdown".

Instance of `MatlabEngine`

`com.mathworks.engine.EngineException`	MATLAB fails to start.

```
MatlabEngine engine = MatlabEngine.startMatlab();
```

“Start and Close MATLAB Session from Java”

startMatlabAsync

```
static Future<MatlabEngine> startMatlabAsync(String[] options)
```

```
static Future<MatlabEngine> startMatlabAsync()
```

Start MATLAB asynchronously

`String[] options`	Startup options used to start MATLAB. For options, see “Startup and Shutdown”.

Instance of `Future<MatlabEngine>`

```
Future<MatlabEngine> future = MatlabEngine.startMatlabAsync();
```

“Start and Close MATLAB Session from Java”

findMatlab

```
static String[] findMatlab()
```

Find all shared MATLAB sessions on the local machine synchronously.

An array of the names of all shared MATLAB sessions on the local machine, or an empty vector if there are no shared MATLAB sessions available on the local machine.

`com.mathworks.engine.EngineException`	If there is a failure during the search for MATLAB sessions.

```
String[] engines = MatlabEngine.findMatlab();
```

“Connect Java to Running MATLAB Session”

`findMatlabAsync`

```
static Future<Sting[]> findMatlabAsync()
```

Find all shared MATLAB sessions on local machine asynchronously.

An instance of `Future<String[]>`

```
Future<String[]> future = MatlabEngine.findMatlabAsync();
```

“Connect Java to Running MATLAB Session”

`connectMatlab`

```
static MatlabEngine connectMatlab(String name)
```

```
static MatlabEngine connectMatlab()
```

Connect to a shared MATLAB session on local machine synchronously.

- If you specify the name of a shared MATLAB session, but the engine cannot find a session with that name, the engine throws an exception.
- If you do not specify a name and there is no shared MATLAB session available, the engine starts a new shared MATLAB session with default options.

- If you do not specify a name and there are shared MATLAB sessions available, the engine connects to the first available session.

`String name`	Name of the shared MATLAB session. Use "findMatlab" on page 1-179 to get the names of shared MATLAB sessions.

An instance of `MatlabEngine`

`com.mathworks.engine.EngineException`	MATLAB fails to start or connect.

```
MatlabEngine engine = MatlabEngine.connectMatlab();
```

"Connect Java to Running MATLAB Session"

connectMatlabAsync

```
static Future<MatlabEngine> connectMatlabAsync(String name)
```

```
static Future<MatlabEngine> connectMatlabAsync
```

Connect to a shared MATLAB session on local machine asynchronously. The behavior is the same as that of `connectMatlab` except the mechanism is asynchronous.

`String name`	Name of the shared MATLAB session.

An instance of `Future<MatlabEngine>`

```
Future<MatlabEngine> future = MatlabEngine.connectMatlabAsync();
```

"Connect Java to Running MATLAB Session"

feval

```
<T> T feval(int nlhs, String func, Writer output, Writer error,
Object… args)
```

```
<T> T feval(int nlhs, String func, Object… args)
```

```
<T> T feval(String func, Writer output, Writer error, Object… args)
```

```
<T> T feval(String func, Object… args)
```

Evaluate MATLAB functions with input arguments synchronously.

`String func`	Name of the MATLAB function or script to evaluate.
`int nlhs`	Number of expected outputs. Default is 1. If `nlhs` is greater than `1`, the returned type `T` must be `<Object[]>`. If `nlhs` is `0`, the returned type `T` must be `<Void>` or `<?>`. If `nlhs` is `1`, the returned type `T` can be the expected type or `<Object>` if the type is not known.
`Writer output`	Stream used to store the standard output from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the output from the MATLAB command window.
`Writer error`	Stream used to store the standard error from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the error message from the MATLAB command window.
`Object... args`	Arguments to pass to the MATLAB function.

Result of executing the MATLAB function

`java.util.concurrent.CancellationException`	Evaluation of a MATLAB function was canceled.

`java.lang.InterruptedException`	Evaluation of a MATLAB function was interrupted.
`java.lang.IllegalStateException`	The MATLAB session is not available.
`com.mathworks.engine.MatlabExcecutionException`	There is a MATLAB runtime error in the function.
`com.mathworks.engine.UnsupportedTypeExeption`	There is an unsupported data type.
`com.mathworks.engine.MatlabSyntaxException`	There is a syntax error in the MATLAB function.

```
double result = engine.feval("sqrt", 4);
```

"Execute MATLAB Functions from Java"

fevalAsync

```
<T> Future<T> fevalAsync(int nlhs, String func, Writer output,
Writer error, Object… args)
```

```
<T> Future<T> fevalAsync(int nlhs, String func, Object… args)
```

```
<T> Future<T> fevalAsync(String func, Writer output, Writer error,
Object… args)
```

```
<T> Future<T> fevalAsync(String func, Object… args)
```

Evaluate MATLAB functions with input arguments asynchronously.

`String func`	Name of the MATLAB function or script to evaluate.

`int nlhs`	Number of expected outputs. Default is `1`. If `nlhs` is greater than `1`, the returned type `T` must be `<Object[]>`. If `nlhs` is `0`, the returned type `T` must be `<Void>` or `<?>`. If `nlhs` is `1`, the returned type `T` can be the expected type or `<Object>` if the type is not known.
`Writer output`	Stream used to store the standard output from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the output from the MATLAB command window.
`Writer error`	Stream used to store the standard error from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the error message from the MATLAB command window.
`Object... args`	Arguments to pass to the MATLAB function.

An instance of `Future<T>`

`java.lang.IllegalStateException`	The MATLAB session is not available.

```
Future<Double> future = engine.fevalAsync("sqrt", 4);
```

“Execute MATLAB Functions from Java”

eval

```
void eval(String command, Writer output, Writer error)
```

```
void eval(String command)
```

Evaluate a MATLAB statement as a string synchronously.

`String command`	MATLAB statement to evaluate.
`Writer output`	Stream used to store the standard output from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the output from the MATLAB command window.
`Writer error`	Stream used to store the standard error from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the error message from the MATLAB command window.

`java.util.concurrent.CancellationException`	Evaluation of a MATLAB function was canceled.
`java.lang.InterruptedException`	Evaluation of a MATLAB function was interrupted.
`java.lang.IllegalStateException`	The MATLAB session is not available.
`com.mathworks.engine.MatlabExcecutionException`	There is an error in the MATLAB statement during runtime.
`com.mathworks.engine.MatlabSyntaxException`	There is a syntax error in the MATLAB statement.

```
engine.eval("result = sqrt(4)");
```

“Evaluate MATLAB Statements from Java”

evalAsync

```
Future<Void> evalAsync(String command, Writer output, Writer error)
```

```
Future<Void> evalAsync(String command)
```

Evaluate a MATLAB statement as a string asynchronously.

`String command`	MATLAB statement to evaluate.

`Writer output`	Stream used to store the standard output from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the output from the MATLAB command window.
`Writer error`	Stream used to store the standard error from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the error message from the MATLAB command window.

An instance of `Future<Void>`

`java.lang.IllegalStateException`	The MATLAB session is not available.

```
Future<Void> future = engine.evalAsync("sqrt(4)");
```

“Evaluate MATLAB Statements from Java”

getVariable

```
<T> T getVariable(String varName)
```

Get a variable from the MATLAB base workspace.

`String varName`	Name of a variable in the MATLAB base workspace.

Variable passed from the MATLAB base workspace

`java.util.concurrent.CancellationException`	Evaluation of this function is canceled.
`java.lang.InterruptedException`	Evaluation of this function is interrupted.

`java.lang.IllegalStateException`	The MATLAB session is not available.

```
double myVar = engine.getVariable("myVar");
```

“Pass Variables from MATLAB to Java”

getVariableAsync

```
<T> Future<T> getVariableAsync(String varName)
```

Get a variable from the MATLAB base workspace asynchronously.

`String varName`	Name of a variable in MATLAB base workspace.

An instance of `Future<T>`

`java.lang.IllegalStateException`	The MATLAB session is not available.

```
Future<Double> future = engine.getVariableAsync("myVar");
```

“Pass Variables from MATLAB to Java”

putVariable

```
void putVariable(String varName, T varData)
```

Put a variable into the MATLAB base workspace.

`String varName`	Name of a variable to create in the MATLAB base workspace.
`T varData`	Value of the variable to create in the MATLAB base workspace.

`java.util.concurrent.CancellationException`	Evaluation of this function is canceled.
`java.lang.InterruptedException`	Evaluation of this function is interrupted.
`java.lang.IllegalStateException`	The MATLAB session is not available.

```
engine.putVariable("myVar", 100);
```

"Pass Variables from Java to MATLAB"

putVariableAsync

```
Future<Void> putVariableAsync(String varName, T varData)
```

Put a variable into the MATLAB base workspace asynchronously.

`String varName`	Name of a variable to create in the MATLAB base workspace.
`T varData`	Value of the variable to create in the MATLAB base workspace.

An instance of `Future<Void>`

`java.lang.IllegalStateException`	The MATLAB session is not available.

```
Future<Void> future = engine.putVariableAsync("myVar", 100);
```

"Pass Variables from Java to MATLAB"

disconnect

```
void disconnect()
```

Disconnect from the current MATLAB session.

`com.mathworks.engine.EngineException`	The current MATLAB session cannot be disconnected.

```
engine.disconnect();
```

“Close MATLAB Engine Session”

disconnectAsync

```
Future<Void> disconnectAsync()
```

Disconnect from the current MATLAB session.

```
Future<Void> future = engine.disconnectAsync();
```

“Close MATLAB Engine Session”

quit

```
viod quit()
```

Force the shutdown of the current MATLAB session.

`com.mathworks.engine.EngineException`	The current MATLAB session cannot be shut down.

```
engine.quit();
```

“Close MATLAB Engine Session”

quitAsync

```
Future<Void> quitAsync()
```

Force the shutdown of the current MATLAB session asynchronously without waiting for termination.

An instance of `Future<Void>`

```
Future<Void> future = engine.quitAsync();
```

“Close MATLAB Engine Session”

close

```
void close()
```

`MatlabEngine` provides the `close()` method to implement the `java.lang.AutoCloseable` interface for `MatlabEngine` objects. This `close()` method enables you to use a `try`-with-resources statement to automatically disconnect or terminate the MATLAB session at the end of the statement.

The `MatlabEngine` `close()` method disconnects or terminates the current MATLAB session, depending on the context.

- If a Java process starts the MATLAB session as a default non-shared session, `close()` terminates MATLAB.
- If the MATLAB session is a shared session, `close()` disconnects MATLAB from this Java process. MATLAB terminates when there are no other connections.

To force the shutdown or disconnection of the current MATLAB session, explicitly call `MatlabEngine.quit()`, `MatlabEngine.disconnect()`, or their asynchronous counterparts.

```
engine.close();
```

“Close MATLAB Engine Session”

Examples

Evaluate Function Asynchronously

This example shows how to evaluate a MATLAB function asynchronously. The workflow is:

- Open a MATLAB session.
- Invoke the MATLAB `sqrt` function with arguments asynchronously.
- Get the result of the MATLAB function.
- Close the MATLAB engine.

```
import com.mathworks.engine.MatlabEngine

Future<MatlabEngine> engFuture = MatlabEngine.startMatlabAsync();
MatlabEngine engine = engFuture.get();
double myVar = 4;
Future<Double> future = engine.fevalAsync("sqrt", myVar);
double result = future.get();
System.out.println(result);
```

See Also

`matlab.engine.engineName` | `matlab.engine.isEngineShared` | `matlab.engine.shareEngine`

Topics

“Build Java Engine Programs”
“Start and Close MATLAB Session from Java”

Introduced in R2016b

com.mathworks.matlab.types.Complex class

Package: com.mathworks.matlab.types

Java class to pass complex data to and from MATLAB

Description

The `Complex` class provides Java support for MATLAB complex arrays. Use this class to pass complex data to MATLAB. The MATLAB engine passes complex data to Java as an instance of `Complex`.

All MATLAB numeric types are converted to `double` in Java.

Constructor Summary

`Complex(double real, double imag)` constructs an instance of `Complex` with the specified real and imaginary values.

Field Summary

`double real`	The real part of the complex data
`double imag`	The imaginary part of the complex data

Examples

Pass Complex Variable to MATLAB Function

```
import com.mathworks.engine.MatlabEngine

MatlabEngine engine = MatlabEngine.startMatlab();
```

```
Complex c = new Complex(2,3);
Complex cj = engine.feval("conj",c);
```

- “Using Complex Variables in Java”

See Also

`com.mathworks.matlab.types.CellStr` | `com.mathworks.matlab.types.HandleObject` | `com.mathworks.matlab.types.Struct`

Topics

“Using Complex Variables in Java”

Introduced in R2016b

com.mathworks.matlab.types.HandleObject class

Package: com.mathworks.matlab.types

Java class to represent MATLAB handle objects

Description

Java represents handle objects that are passed from MATLAB as instances of the `HandleObject` class. When passing a handle object back to MATLAB, Java passes a reference to the `HandleObject` instance. This reference can be either an array or a scalar, depending on the original handle object passed to Java from MATLAB.

You can pass a handle object only to the MATLAB session in which it was originally created. You cannot construct a `HandleObject` in Java.

Examples

Get Handle Object from MATLAB

This example starts a shared MATLAB session and creates a `containers.Map` object in the MATLAB workspace. The statement evaluated in the MATLAB workspace returns a handle variable that refers to the `Map` object.

The engine `getVariable` function returns the MATLAB handle variable as a `HandleObject` instance. This instance is used to call the MATLAB `containers.Map.keys` function to obtain the `Map` keys.

```
import com.mathworks.engine.MatlabEngine;
import com.mathworks.matlab.types.*;

MatlabEngine engine = MatlabEngine.startMatlab();
engine.eval("cm = containers.Map({'id','name'},{11,'mw'});");
```

```
HandleObject handle = engine.getVariable("cm");
String[] cells = engine.feval("keys", handle);
```

See Also

`com.mathworks.matlab.types.CellStr` | `com.mathworks.matlab.types.Complex` | `com.mathworks.matlab.types.Struct`

Topics

"Using MATLAB Handle Objects in Java"

Introduced in R2016b

com.mathworks.matlab.types.Struct class

Package: com.mathworks.matlab.types

Java class to pass MATLAB `struct` to and from MATLAB

Description

The `Struct` class provides support for passing data between MATLAB and Java as a MATLAB `struct`. The `Struct` class implements the `java.util.Map` interface.

The `Struct` class is designed as an immutable type. Attempting to change the mappings, keys, or values of the returned `Struct` causes an `UnsupportedOperationException`. Calling these methods can cause the exception: `put()`, `putAll()`, `remove()`, `entrySet()`, `keySet()`, and `values()`.

For an example, see "Using MATLAB Structures in Java".

Constructor Summary

`Struct s = new Struct("field1",value1,"field2",value2, ...)` creates an instance of `Struct` with the specified field names and values.

Method Summary

`containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.
`entrySet()`	Returns a `Set` view of the mappings contained in this map.
`equals(Object o)`	Compares the specified object with this map for equality.

`get(Object key)`	Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`hashCode()`	Returns the hash code value for this map.
`isEmpty()`	Returns `true` if this map contains no key-value mappings.
`keySet()`	Returns a `Set` view of the keys contained in this map.
`size()`	Returns the number of key-value mappings in this map.
`values()`	Returns a `Collection` view of the values contained in this map.

Examples

Create `Struct` for MATLAB Function Argument

Create a `Struct` and assign a key and value.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

class StructProperties {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        int[] y = {1,2,3,4,5};
        double[] color = {1.0,0.5,0.7};
        Struct s = new Struct("Color",color,"LineWidth",2);
        eng.feval("plot",y,s);
    }
}
```

- “Using MATLAB Structures in Java”

See Also

`com.mathworks.matlab.types.CellStr` | `com.mathworks.matlab.types.Complex` | `com.mathworks.matlab.types.HandleObject`

Topics

"Using MATLAB Structures in Java"

Introduced in R2016b

com.mathworks.matlab.types.CellStr class

Package: com.mathworks.matlab.types

Java class to represent MATLAB cell array of `char` vectors

Description

The `CellStr` class provides support for passing data from Java to MATLAB as a MATLAB cell array of `char` vectors (called a `cellstr` in MATLAB, see `cellstr`). There are MATLAB functions that require cell arrays of `char` vectors as inputs. To pass arguments from Java to a MATLAB function requiring `cellst` inputs, use the Java `CellStr` class to create a compatible type.

A MATLAB `cellstr` is mapped to a Java `String` array.

Constructor Summary

`CellStr(Object stringArray)` creates a `CellStr` using a `String` or `String` array. The `String` array can have multiple dimensions.

Method Summary

`Object getStringArray()`	Get the `String` or `String` array used to create the `CellStr`.
`boolean equals(CellStr1,CellStr2)`	Compare one `CellStr` instance with another. Two `CellStr` instances are equal if the `String` or `String` array they contain are the same.

Examples

Construct CellStr

This example constructs a `CellStr` named `keySet` and puts the variable in the MATLAB base workspace.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

class javaCellstr {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        CellStr keySet = new CellStr(new String[]{"Jan","Feb","Mar","Apr"});
        eng.putVariable("mapKeys",keySet);
        eng.close();
    }
}
```

Construct CellStr Array

This example creates a `CellStr` array and passes it to the MATLAB `plot` function to change the appearance of the graph produced by MATLAB. The call to the MATLAB `print` function exports the figure as a `jpeg` file named `myPlot.jpg`.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

class CellStrArray {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        String[][] strArray = new String[2][2];
        strArray[0][0] = "MarkerFaceColor";
        strArray[0][1] = "MarkerEdgeColor";
        strArray[1][0] = "green";
        strArray[1][1] = "red";
        CellStr markerCellStr = new CellStr(strArray);
        eng.putVariable("M",markerCellStr);
        eng.eval("plot(1:10,'--bs',M{:})");
        eng.eval("print('myPlot','-djpeg')");
        eng.close();
    }
}
```

See Also

`com.mathworks.matlab.types.Complex` | `com.mathworks.matlab.types.HandleObject` | `com.mathworks.matlab.types.Struct`

Topics

"Pass Java CellStr to MATLAB"

Introduced in R2016b

engClose (C and Fortran)

Quit MATLAB engine session

C Syntax

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

Fortran Syntax

```
#include "engine.h"
integer*4 engClose(ep)
mwPointer ep
```

Arguments

`ep`

Engine pointer

Returns

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

Description

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

Examples

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

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

See Also

engOpen

Introduced before R2006a

engEvalString (C and Fortran)

Evaluate expression in string

C Syntax

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

Fortran Syntax

```
#include "engine.h"
integer*4 engEvalString(ep, string)
mwPointer ep
character*(*) string
```

Arguments

`ep`

Engine pointer

`string`

String to execute

Returns

`1` if the engine session is no longer running or the engine pointer is invalid or `NULL`. Otherwise, returns `0` even if the MATLAB engine session cannot evaluate the command.

Description

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

UNIX Operating Systems

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

To turn off output buffering in C, use:

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

To turn off output buffering in Fortran, use:

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

Microsoft Windows Operating Systems

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

Examples

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

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

See Also

`engOpen`, `engOutputBuffer`

Introduced before R2006a

engGetVariable (C and Fortran)

Copy variable from MATLAB engine workspace

C Syntax

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

Fortran Syntax

```
#include "engine.h"
mwPointer engGetVariable(ep, name)
mwPointer ep
character*(*) name
```

Arguments

`ep`

Engine pointer

`name`

Name of `mxArray` to get from MATLAB workspace

Returns

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

Description

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

The limit for the size of data transferred is 2 GB.

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

Examples

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

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

See Also

`engPutVariable`, `mxDestroyArray`

Introduced before R2006a

engGetVisible (C)

Determine visibility of MATLAB engine session

C Syntax

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

Arguments

`ep`

Engine pointer

`value`

Pointer to value returned from `engGetVisible`

Returns

Microsoft Windows Operating Systems Only

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

Description

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

Examples

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

```
Engine *ep;
bool vis;

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

To determine the current visibility setting, use:

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

See Also

engSetVisible

Engine (C)

Type for MATLAB engine

Description

A handle to a MATLAB engine object.

`Engine` is a C language opaque type.

You can call MATLAB software as a computational engine by writing C and Fortran programs that use the MATLAB engine library. `Engine` is the link between your program and the separate MATLAB engine process.

The header file containing this type is:

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

Examples

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

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

See Also

`engOpen`

engOpen (C and Fortran)

Start MATLAB engine session

C Syntax

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

Fortran Syntax

```
#include "engine.h"
mwPointer engOpen(startcmd)
character*(*) startcmd
```

Arguments

`startcmd`

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

Returns

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

Description

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

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

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

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

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

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

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

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

```
!matlab -regserver
```

See "MATLAB COM Integration" for additional details.

Examples

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

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

Introduced before R2006a

engOpenSingleUse (C)

Start MATLAB engine session for single, nonshared use

C Syntax

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

Arguments

`startcmd`

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

`dcom`

Reserved for future use; must be `NULL`.

`retstatus`

Return status; possible cause of failure.

Returns

Microsoft Windows Operating Systems Only

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

UNIX Operating Systems

Not supported on UNIX systems.

Description

This routine allows you to start multiple MATLAB processes using MATLAB as a computational engine.

`engOpenSingleUse` starts a MATLAB process, establishes a connection, and returns a unique engine identifier, or `NULL` if the open fails. Each call to `engOpenSingleUse` starts a new MATLAB process.

`engOpenSingleUse` opens a COM channel to MATLAB. This starts the MATLAB software you registered during installation. If you did not register during installation, enter the following command at the MATLAB prompt:

```
!matlab -regserver
```

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

See “MATLAB COM Integration” for additional details.

engOutputBuffer (C and Fortran)

Specify buffer for MATLAB output

C Syntax

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

Fortran Syntax

```
#include "engine.h"
integer*4 engOutputBuffer(ep, p)
mwPointer ep
character*n p
```

Arguments

`ep`

Engine pointer

`p`

Pointer to character buffer

`n`

Length of buffer `p`

Returns

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

Description

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

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

To turn off output buffering in C, use:

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

To turn off output buffering in Fortran, use:

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

Note The buffer returned by engEvalString is not NULL terminated.

Examples

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

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

See Also

engOpen, engEvalString

Introduced before R2006a

engPutVariable (C and Fortran)

Put variable into MATLAB engine workspace

C Syntax

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

Fortran Syntax

```
#include "engine.h"
integer*4 engPutVariable(ep, name, pm)
mwPointer ep, pm
character*(*) name
```

Arguments

`ep`

Engine pointer

`name`

Name of `mxArray` in the engine workspace

`pm`

`mxArray` pointer

Returns

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

Description

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

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

The limit for the size of data transferred is 2 GB.

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

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

Examples

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

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

See Also

engGetVariable

Introduced before R2006a

engSetVisible (C)

Show or hide MATLAB engine session

C Syntax

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

Arguments

`ep`

Engine pointer

`value`

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

Returns

Microsoft Windows Operating Systems Only

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

Description

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

Examples

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

```
Engine *ep;
bool vis;

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

To determine the current visibility setting, use:

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

See Also

`engGetVisible`

matClose (C and Fortran)

Close MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
integer*4 matClose(mfp)
mwPointer mfp
```

Arguments

`mfp`

Pointer to MAT-file information

Returns

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

Description

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

Examples

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

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

See Also

`matOpen`

Introduced before R2006a

matDeleteVariable (C and Fortran)

Delete array from MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
integer*4 matDeleteVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to delete

Returns

`0` if successful, and nonzero otherwise.

Description

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

Examples

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

- matdemo1.F

Introduced before R2006a

MATFile (C and Fortran)

Type for MAT-file

Description

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

`MATFile` is a C language opaque type.

The MAT-file interface library contains routines for reading and writing MAT-files. Call these routines from your own C/C++ and Fortran programs, using `MATFile` to access your data file.

The header file containing this type is:

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

Examples

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

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

See Also

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

matGetDir (C and Fortran)

List of variables in MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
mwPointer matGetDir(mfp, num)
mwPointer mfp
integer*4 num
```

Arguments

`mfp`

Pointer to MAT-file information

`num`

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

Returns

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

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

Description

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

`matGetDir` allocates memory for the internal array of strings using a `mxCalloc`. Free the memory using `mxFree` when you are finished with the array.

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

Examples

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

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

Introduced before R2006a

matGetFp (C)

File pointer to MAT-file

C Syntax

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

Arguments

`mfp`

Pointer to MAT-file information

Returns

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

Description

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

Introduced before R2006a

matGetNextVariable (C and Fortran)

Next array in MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
mwPointer matGetNextVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Pointer to the variable containing the `mxArray` name

Returns

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

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

Description

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

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

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

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

Examples

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

- `matdgns.c`
- `matdemo2.F`

See Also

`matGetNextVariableInfo`, `matGetVariable`, `mxDestroyArray`

Introduced before R2006a

matGetNextVariableInfo (C and Fortran)

Array header information only

C Syntax

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

Fortran Syntax

```
#include "mat.h"
mwPointer matGetNextVariableInfo(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Pointer to the variable containing the `mxArray` name

Returns

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

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

Description

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

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

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

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

Examples

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

- `matdgns.c`
- `matdemo2.F`

See Also

`matGetNextVariable`, `matGetVariableInfo`

Introduced before R2006a

matGetVariable (C and Fortran)

Array from MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
mwPointer matGetVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to get from MAT-file

Returns

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

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

Description

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

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

Examples

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

- `matcreat.c`
- `matdemo1.F`

See Also

`matPutVariable`, `mxDestroyArray`

Introduced before R2006a

matGetVariableInfo (C and Fortran)

Array header information only

C Syntax

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

Fortran Syntax

```
#include "mat.h"
mwPointer matGetVariableInfo(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to get from MAT-file

Returns

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

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

Description

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

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

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

Examples

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

- matdemo2.F

See Also

matGetVariable

Introduced before R2006a

matlab.engine.connect_matlab

Connect shared MATLAB session to MATLAB Engine for Python

Syntax

```
eng = matlab.engine.connect_matlab(name=None)
eng = matlab.engine.connect_matlab(___,async)
```

Description

`eng = matlab.engine.connect_matlab(name=None)` connects to the shared MATLAB session, `name`, and returns a `MatlabEngine` object as `eng`. The input argument `name` specifies the name of a MATLAB session that is already running on your local machine.

- If you specify `name` and the engine cannot find a shared MATLAB session of the same name, then you receive an `EngineError` exception.
- If you do not specify `name` and the engine cannot find any shared MATLAB sessions, then it starts a new shared MATLAB session. If there are shared MATLAB sessions running, the engine connects to the first available session.
- If you do not specify `name` and the engine finds multiple shared MATLAB sessions running, then it connects to the first available session.

`eng = matlab.engine.connect_matlab(___,async)` connects asynchronously if `async` is `True`.

Examples

Connect to MATLAB Session

Connect to a shared MATLAB session that is already running on your local machine.

```
import matlab.engine
eng = matlab.engine.connect_matlab()
eng.sqrt(4.0)
```

```
2.0
```

`matlab.engine.connect_matlab` connects to the first available shared MATLAB session. If no MATLAB sessions are shared, `matlab.engine.connect_matlab` starts a new session.

Connect to MATLAB Sessions by Name

When there are multiple shared MATLAB sessions on your local machine, connect to two different sessions one at a time by specifying their names.

Connect to the first shared MATLAB session.

```
import matlab.engine
names = matlab.engine.find_matlab()
names
```

```
('MATLAB_6830', 'MATLAB_7090')
```

Connect to the second shared MATLAB session.

```
eng = matlab.engine.connect_matlab('MATLAB_7090')
eng.sqrt(4.0)
```

```
2.0
```

- “Connect Python to Running MATLAB Session”

Input Arguments

`name` — Name of shared MATLAB session
character array

Name of the shared MATLAB session, specified as a character array.

`async` — Start MATLAB synchronously or asynchronously
`False` (default) | logical

Start MATLAB synchronously or asynchronously, specified as a logical keyword argument.

Example: `matlab.engine.start_matlab(async=True)`

Output Arguments

`eng` — Python® variable for communicating with MATLAB
`MatlabEngine` object

Python variable for communicating with MATLAB, returned as a `MatlabEngine` object. `eng` communicates with a shared MATLAB session that is already running on your local machine

Limitations

- Do not connect the engine multiple times to the same shared MATLAB session.

See Also

`matlab.engine.MatlabEngine` | `matlab.engine.find_matlab`

Topics

"Connect Python to Running MATLAB Session"

Introduced in R2015b

matlab.engine.find_matlab

Find shared MATLAB sessions to connect to MATLAB Engine for Python

Syntax

```
names = matlab.engine.find_matlab()
```

Description

`names = matlab.engine.find_matlab()` finds all shared MATLAB sessions on your local machine and returns their names in a `tuple`. Any name in `names` can be the input argument to `matlab.engine.connect_matlab`. If there are no shared sessions running on your local machine, `matlab.engine.find_matlab` returns an empty `tuple`.

Examples

Find Shared MATLAB Sessions

Identify the shared MATLAB sessions running on your local machine and connect to one of them.

```
import matlab.engine
names = matlab.engine.find_matlab()
names
```

```
('MATLAB_6830', 'MATLAB_7090')
```

There are two shared MATLAB sessions running, so `matlab.engine.find_matlab` returns two names in a `tuple`.

Connect to the first shared MATLAB session.

```
eng = matlab.engine.connect_matlab('MATLAB_6830')
```

- “Connect Python to Running MATLAB Session”

See Also

`matlab.engine.connect_matlab`

Topics

“Connect Python to Running MATLAB Session”

Introduced in R2015b

matlab.engine.FutureResult class

Package: matlab.engine

Results of asynchronous call to MATLAB function stored in Python object

Description

The `FutureResult` class stores results of an asynchronous call to a MATLAB function in a Python object.

Construction

The MATLAB Engine for Python creates a `FutureResult` object when a MATLAB function is called asynchronously. There is no need to call `matlab.engine.FutureResult()` to create `FutureResult` objects of your own.

Methods

cancel	Cancel asynchronous call to MATLAB function from Python
cancelled	Cancellation status of asynchronous call to MATLAB function from Python
done	Completion status of asynchronous call to MATLAB function from Python
result	Result of asynchronous call to MATLAB function from Python

Exceptions

`SyntaxError`	Python exception, syntax error in function call
`TypeError`	Python exception, data type of output argument not supported

`matlab.engine.CancelledError`	MATLAB engine cannot cancel function call
`matlab.engine.InterruptedError`	Function call interrupted
`matlab.engine.MatlabExecutionError`	Function call fails to execute
`matlab.engine.RejectedExecutionError`	Engine terminated
`matlab.engine.TimeoutError`	Result cannot be returned within the timeout period

Examples

Get Result of Asynchronous MATLAB Call from Python

Call the MATLAB `sqrt` function from Python. Set `async` to `True` to make the function call asynchronously.

```
import matlab.engine
eng = matlab.engine.start_matlab()
future = eng.sqrt(4.0,async=True)
ret = future.result()
print(ret)
```

```
2.0
```

- “Call MATLAB Functions from Python”
- “Call MATLAB Functions Asynchronously from Python”

See Also

`matlab.engine.MatlabEngine`

Topics

“Call MATLAB Functions from Python”
“Call MATLAB Functions Asynchronously from Python”

Introduced in R2014b

cancel

Class: matlab.engine.FutureResult
Package: matlab.engine

Cancel asynchronous call to MATLAB function from Python

Syntax

```
tf = FutureResult.cancel()
```

Description

`tf = FutureResult.cancel()` cancels a call to a MATLAB function called asynchronously from Python. `FutureResult.cancel` returns `True` if it successfully cancels the function, and `False` if it cannot cancel the function.

Output Arguments

`tf` — Cancellation status

`True` | `False`

Cancellation status, returned as either `True` or `False`. The status, `tf`, is `True` if `FutureResult.cancel` successfully cancels the asynchronous function call, and is `False` otherwise.

Examples

Cancel an Asynchronous Call

Start an endless loop in MATLAB with an asynchronous call to the `eval` function. Then, cancel it.

```
import matlab.engine
eng = matlab.engine.start_matlab()
ret = eng.eval("while 1; end",nargout=0,async=True)
tf = ret.cancel()
print(tf)
```

```
True
```

See Also

cancelled

Class: matlab.engine.FutureResult
Package: matlab.engine

Cancellation status of asynchronous call to MATLAB function from Python

Syntax

```
tf = FutureResult.cancelled()
```

Description

`tf = FutureResult.cancelled()` returns the cancellation status of a call to a MATLAB function called asynchronously from Python. `FutureResult.cancelled` returns `True` if a previous call to `FutureResult.cancel` succeeded, and `False` otherwise.

Output Arguments

`tf` — Cancellation status

`True` | `False`

Cancellation status of an asynchronous function call, returned as either `True` or `False`.

Examples

Check Cancellation Status of Asynchronous Call

Start an endless loop in MATLAB with an asynchronous call to the `eval` function. Cancel it and check that the engine stopped the loop.

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

```
ret = eng.eval("while 1; end",nargout=0,async=True)
eval_stop = ret.cancel()
tf = ret.cancelled()
print(tf)
```

```
True
```

See Also

done

Class: matlab.engine.FutureResult
Package: matlab.engine

Completion status of asynchronous call to MATLAB function from Python

Syntax

```
tf = FutureResult.done()
```

Description

`tf = FutureResult.done()` returns the completion status of a MATLAB function called asynchronously from Python. `FutureResult.done` returns `True` if the function has finished, and `False` if it has not finished.

Output Arguments

`tf` — Completion status of asynchronous function call

`True` | `False`

Completion status of an asynchronous function call, returned as either `True` or `False`.

Examples

Check If Asynchronous Call Finished

Call the MATLAB `sqrt` function with `async = True`. Check the status of `ret` to learn if `sqrt` is finished.

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

```
ret = eng.sqrt(4.0,async=True)
tf = ret.done()
print(tf)
```

```
True
```

When `ret.done()` returns `True`, then you can call `ret.result()` to return the square root.

See Also

result

Class: matlab.engine.FutureResult
Package: matlab.engine

Result of asynchronous call to MATLAB function from Python

Syntax

```
ret = FutureResult.result(timeout=None)
```

Description

`ret = FutureResult.result(timeout=None)` returns the actual result of a call to a MATLAB function called asynchronously from Python.

Input Arguments

`timeout` — Timeout value in seconds

`None` (default) | Python `float`

Timeout value in seconds, specified as Python data type `float`, to wait for result of the function call. If `timeout = None`, the `FutureResult.result` function waits until the function call finishes, and then returns the result.

Output Arguments

`ret` — Result of asynchronous function call

Python object

Result of an asynchronous function call, returned as a Python object, that is the actual output argument of a call to a MATLAB function.

Examples

Get MATLAB Output Argument from Asynchronous Call

Call the MATLAB `sqrt` function from Python. Set `async` to `True` and get the square root from the `FutureResult` object.

```
import matlab.engine
eng = matlab.engine.start_matlab()
future = eng.sqrt(4.0,async=True)
ret = future.result()
print(ret)
```

```
2.0
```

See Also

matlab.engine.MatlabEngine class

Package: matlab.engine

Python object using MATLAB as computational engine within Python session

Description

The `MatlabEngine` class uses a MATLAB process as a computational engine for Python. You can call MATLAB functions as methods of a `MatlabEngine` object because the functions are dynamically invoked when you call them. You also can call functions and scripts that you define. You can send data to, and retrieve data from, the MATLAB workspace associated with a `MatlabEngine` object.

Construction

The `matlab.engine.start_matlab` function creates a `MatlabEngine` object each time it is called. There is no need to call `matlab.engine.MatlabEngine()` to create `MatlabEngine` objects of your own.

Methods

You can call any MATLAB function as a method of a `MatlabEngine` object. The engine dynamically invokes a MATLAB function when you call it. The syntax shows positional, keyword, and output arguments of a function call.

```
ret =
MatlabEngine.matlabfunc(*args,nargout=1,async=False,stdout=sys.stsdo
ut,stderr=sys.stderr)
```

Replace *`matlabfunc`* with the name of any MATLAB function (such as `isprime` or `sqrt`). Replace *`*args`* with input arguments for the MATLAB function you call. The keyword arguments specify:

- The number of output arguments the function returns
- Whether the engine calls the function asynchronously
- Where the engine sends standard output and standard error coming from the function

Specify keyword arguments only when specifying values that are not the default values shown in the syntax.

Input Arguments to MATLAB Function

Argument	Description	Python Type
`*args`	Input arguments to MATLAB function, specified as positional arguments	Any Python types that the engine can convert to MATLAB types

Keyword Arguments to Engine

Argument	Description	Python Type
`nargout`	Number of output arguments from MATLAB function	`int` **Default:** `1`
`async`	Flag to call MATLAB function asynchronously	`bool` **Default:** `False`
`stdout`	Standard output	`StringIO.StringIO` object (Python 2.7) `io.StringIO` object (Python 3.x) **Default:** `sys.stdout`
`stderr`	Standard error	`StringIO.StringIO` object (Python 2.7) `io.StringIO` object (Python 3.x) **Default:** `sys.stderr`

Output Arguments

Output Type	Description	Required Keyword Arguments
Python variable	One output argument from MATLAB function	Default values
`tuple`	Multiple output arguments from MATLAB function	`nargout=`*n* (where *n* `> 1`)
`None`	No output argument from MATLAB function	`nargout=0`
`FutureResult` object	A placeholder for output arguments from asynchronous call to MATLAB function	`async=True`

Exceptions

`MatlabExecutionError`	Function call fails to execute
`RejectedExecutionError`	MATLAB engine terminated
`SyntaxError`	Syntax error in a function call
`TypeError`	Data type of an input or output argument not supported

Attributes

`workspace`	Python dictionary containing references to MATLAB variables. You can assign data to, and get data from, a MATLAB variable through the `workspace`. The name of each MATLAB variable you create becomes a key in the `workspace` dictionary. The keys in `workspace` must be valid MATLAB identifiers (for example, you cannot use numbers as keys).

Examples

Call MATLAB Functions from Python

Call the MATLAB `sqrt` function from Python using the engine.

```
import matlab.engine
eng = matlab.engine.start_matlab()
ret = eng.sqrt(4.0)
print(ret)
```

```
2.0
```

Put Array Into MATLAB Workspace

Create an array in Python and put it into the MATLAB workspace.

```
import matlab.engine
eng = matlab.engine.start_matlab()
px = eng.linspace(0.0,6.28,1000)
```

`px` is a MATLAB array, but `eng.linspace` returned it to Python. To use it in MATLAB, put the array into the MATLAB workspace.

```
eng.workspace['mx'] = px
```

When you add an entry to the engine `workspace` dictionary, you create a MATLAB variable, as well. The engine converts the data to a MATLAB data type.

Get Data from MATLAB Workspace

Get `pi` from the MATLAB workspace and copy it to a Python variable.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.eval('a = pi;',nargout=0)
mpi = eng.workspace['a']
print(mpi)
```

```
3.14159265359
```

- "Call MATLAB Functions from Python"
- "Call MATLAB Functions Asynchronously from Python"
- "Redirect Standard Output and Error to Python"

See Also

`matlab.engine.FutureResult` | `matlab.engine.start_matlab`

Topics

"Call MATLAB Functions from Python"
"Call MATLAB Functions Asynchronously from Python"
"Redirect Standard Output and Error to Python"

Introduced in R2014b

matlab.engine.start_matlab

Start MATLAB Engine for Python

Syntax

```
eng = matlab.engine.start_matlab()

eng = matlab.engine.start_matlab(option)
eng = matlab.engine.start_matlab(async)
eng = matlab.engine.start_matlab(background)
eng = matlab.engine.start_matlab( ___ )
```

Description

`eng = matlab.engine.start_matlab()` starts a new MATLAB process, and returns Python variable `eng`, which is a `MatlabEngine` object for communicating with the MATLAB process.

If MATLAB cannot be started, the engine raises an `EngineError` exception.

`eng = matlab.engine.start_matlab(option)` uses startup options specified by `option`.

For example, call `matlab.engine.start_matlab('-desktop')` to start the MATLAB desktop from Python.

`eng = matlab.engine.start_matlab(async)` starts MATLAB asynchronously if `async` is `True`.

`eng = matlab.engine.start_matlab(background)` starts MATLAB asynchronously if `background` is `True`.

`eng = matlab.engine.start_matlab( ___ )` can include any of the input arguments in previous syntaxes.

Examples

Start MATLAB Engine from Python

Start an engine and a new MATLAB process from the Python command line.

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

Start Multiple Engines

Start a different MATLAB process from each engine.

```
import matlab.engine
eng1 = matlab.engine.start_matlab()
eng2 = matlab.engine.start_matlab()
```

Start MATLAB Desktop with Engine

Start an engine with the MATLAB desktop.

```
import matlab.engine
eng = matlab.engine.start_matlab("-desktop")
```

You also can start the desktop after you start the engine.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.desktop(nargout=0)
```

Note You can call MATLAB functions from both the desktop and Python.

Start Engine Asynchronously

Start the engine with `async=True`. While MATLAB starts, you can enter commands at the Python command line.

```
import matlab.engine
future = matlab.engine.start_matlab(async=True)
eng = future.result()
eng.sqrt(4.)
```

```
2.0
```

- “Start and Stop MATLAB Engine for Python”

Input Arguments

`option` — Startup options for MATLAB process
`'-nodesktop'` (default) | string

Startup options for the MATLAB process, specified as a string. You can specify multiple startup options with `option`.

The engine supports `'-desktop'` to start MATLAB with the desktop. In addition, the engine supports all MATLAB startup options, except for the options listed in “Limitations” on page 1-261.

Example: `matlab.engine.start_matlab('-desktop -r "format short"')` starts the desktop from Python. The engine passes `'-r "format short"'` to MATLAB.

`async` — Start MATLAB synchronously or asynchronously
`False` (default) | logical

Start MATLAB synchronously or asynchronously, specified as a logical keyword argument.

Example: `matlab.engine.start_matlab(async=True)`

`background` — Start MATLAB synchronously or asynchronously
`False` (default) | logical

Start MATLAB synchronously or asynchronously, specified as a logical keyword argument. `background` is an alias for `async` and will be removed in a future release.

Example: `matlab.engine.start_matlab(background=True)`

Output Arguments

eng — Python variable for communicating with MATLAB
MatlabEngine object | FutureResult object

Python variable for communicating with MATLAB, returned as a MatlabEngine object if async or background is set to False or a FutureResult object if async or background is set to True.

Each time you call matlab.engine.start_matlab, it starts a new MATLAB process.

Limitations

The engine does not support these MATLAB startup options:

- -h
- -help
- -?
- -n
- -e
- -softwareopengl
- -logfile

See Also

matlab.engine.MatlabEngine

Topics

"Start and Stop MATLAB Engine for Python"
"Specify Startup Options"
"Commonly Used Startup Options"

Introduced in R2014b

matOpen (C and Fortran)

Open MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
mwPointer matOpen(filename, mode)
character*(*) filename, mode
```

Arguments

`filename`

Name of file to open

`mode`

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

`r`	Opens file for reading only; determines the current version of the MAT-file by inspecting the files and preserves the current version.
`u`	Opens file for update, both reading and writing. If the file does not exist, does not create a file (equivalent to the `r+` mode of `fopen`). Determines the current version of the MAT-file by inspecting the files and preserves the current version.
`w`	Opens file for writing only; deletes previous contents, if any.
`w4`	Creates a MAT-file compatible with MATLAB Versions 4 software and earlier.
`w6`	Creates a MAT-file compatible with MATLAB Version 5 (R8) software or earlier. Equivalent to `wL` mode.

wL	Opens file for writing character data using the default character set for your system. Use MATLAB Version 6 or 6.5 software to read the resulting MAT-file. If you do not use the wL mode switch, MATLAB writes character data to the MAT-file using Unicode® character encoding by default. Equivalent to w6 mode.
w7	Creates a MAT-file compatible with MATLAB Version 7.0 (R14) software or earlier. Equivalent to wz mode.
wz	Opens file for writing compressed data. By default, the MATLAB save function compresses workspace variables as they are saved to a MAT-file. To use the same compression ratio when creating a MAT-file with the matOpen function, use the wz option. Equivalent to w7 mode.
w7.3	Creates a MAT-file in an HDF5-based format that can store objects that occupy more than 2 GB.

Returns

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

Description

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

Examples

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

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

- `matdemo2.F`

See Also

`matClose`, `save`

Introduced before R2006a

matPutVariable (C and Fortran)

Array to MAT-file

C Syntax

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

Fortran Syntax

```
#include "mat.h"
integer*4 matPutVariable(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to put into MAT-file

`pm`

`mxArray` pointer

Returns

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

Description

This routine puts an `mxArray` into a MAT-file.

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

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

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

Examples

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

- `matcreat.c`
- `matdemo1.F`

See Also

`matGetVariable`, `matGetFp`

Introduced before R2006a

matPutVariableAsGlobal (C and Fortran)

Array to MAT-file as originating from global workspace

C Syntax

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

Fortran Syntax

```
#include "mat.h"
integer*4 matPutVariableAsGlobal(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to put into MAT-file

`pm`

`mxArray` pointer

Returns

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

Description

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

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

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

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

Examples

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

- matcreat.c
- matdemo1.F

See Also

matPutVariable, matGetFp

Introduced before R2006a

mexAtExit (C and Fortran)

Register function to call when MEX function clears or MATLAB terminates

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mexAtExit(ExitFcn)
subroutine ExitFcn()
```

Arguments

`ExitFcn`

Pointer to function you want to run on exit

Returns

Always returns `0`.

Description

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

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

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

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

Examples

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

- `mexatexit.c`

See Also

`mexLock`, `mexUnlock`

Introduced before R2006a

mexCallMATLAB (C and Fortran)

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

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mexCallMATLAB(nlhs, plhs, nrhs, prhs, functionName)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) functionName
```

Arguments

`nlhs`

Number of output arguments. Must be less than or equal to 50.

`plhs`

Array of pointers to output arguments

`nrhs`

Number of input arguments. Must be less than or equal to 50.

`prhs`

Array of pointers to input arguments

`functionName`

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

If `functionName` is an operator, 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, user-defined functions, or other MEX files. Both `mexCallMATLAB` and `mexEvalString` execute MATLAB commands. Use `mexCallMATLAB` for returning results (left side arguments) back to the MEX file. The `mexEvalString` function cannot return values to the MEX file.

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

Error Handling

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

Limitations

- Avoid using the `mexCallMATLAB` function in Simulink® S-functions. If you do, do not store the resulting `plhs` `mxArray` pointers in any S-function block state that persists after the MEX function finishes. Outputs of `mexCallMATLAB` have temporary scope and are automatically destroyed at the end of the MEX function call.
- It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, this function returns two variables but only assigns one of them a value.

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

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

Examples

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

- `mexcallmatlab.c`
- `mexevalstring.c`
- `mexcallmatlabwithtrap.c`

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

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

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

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

See Also

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

Tips

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

  Note The `plhs` argument for `mexCallMATLAB` is not the same as the `plhs` for `mexFunction`. Do not destroy an `mxArray` returned in `plhs` for `mexFunction`.

Introduced before R2006a

mexCallMATLABWithTrap (C and Fortran)

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

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mexCallMATLABWithTrap(nlhs, plhs, nrhs, prhs, functionName)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) functionName
```

Arguments

For more information about arguments, see `mexCallMATLAB`.

`nlhs`

Number of desired output arguments.

`plhs`

Array of pointers to output arguments.

`nrhs`

Number of input arguments.

`prhs`

Array of pointers to input arguments.

`functionName`

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

Returns

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

Description

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

See Also

`mexCallMATLAB`, `MException`

Introduced in R2008b

mexErrMsgIdAndTxt (C and Fortran)

Display error message with identifier and return to MATLAB prompt

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexErrMsgIdAndTxt(errorid, errormsg)
character*(*) errorid, errormsg
```

Arguments

`errorid`

String containing a MATLAB message identifier. For information on creating identifiers, see “Message Identifiers”.

`errormsg`

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

`...`

In C, any arguments used in the message. Each argument must have a corresponding conversion specification. Refer to your C documentation for `printf` conversion tables.

Description

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

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

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

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

Remarks

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

```
Error using foo
```

If you compile your MEX file with the MinGW-w64 compiler, see the limitations with exception handling topic in “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64”.

Examples

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

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

Validate `char` Input

The following code snippet checks if input argument, `prhs[0]`, is a string. If not, the code displays a warning. If there is an error reading the input string, the code displays an error message and terminates the MEX file.

```
char *buf;
int buflen;

if (mxIsChar(prhs[0])) {
    if (mxGetString(prhs[0], buf, buflen) == 0) {
        mexPrintf("The input string is:  %s\n", buf);
    }
    else {
        mexErrMsgIdAndTxt("MyProg:ConvertString",
           "Could not convert string data.");
        // exit MEX file
    }
}
else {
    mexWarnMsgIdAndTxt("MyProg:InputString",
        "Input should be a string to print properly.");
}

// continue with processing
```

See Also

`error` | `mexWarnMsgIdAndTxt`

Topics

"Memory Considerations for Class Destructors"
"Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64"

Introduced before R2006a

mexErrMsgTxt (C and Fortran)

Display error message and return to MATLAB prompt

Note `mexErrMsgTxt` is not recommended. Use `mexErrMsgIdAndTxt` instead.

C Syntax

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

Fortran Syntax

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

Arguments

`errormsg`

String containing the error message to display

Description

`mexErrMsgTxt` writes 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. So, `mexErrMsgTxt` does not invoke the function registered through `mexAtExit`.

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

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

Remarks

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

```
Error using foo
Print my error message
```

See Also

`mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

mexEvalString (C and Fortran)

Execute MATLAB command in caller workspace

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mexEvalString(command)
character*(*) command
```

Arguments

`command`

String containing MATLAB command to execute

Returns

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

Description

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

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. Use `mexCallMATLAB` for returning results (left side arguments) back to the MEX file. The `mexEvalString` function cannot return values to the MEX file.

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

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

Error Handling

If `command` detects an error, MATLAB returns control to the MEX-file and `mexEvalString` returns `1`. If you want to trap errors, use the `mexEvalStringWithTrap` function.

Examples

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

- `mexevalstring.c`

See Also

`mexCallMATLAB`, `mexEvalStringWithTrap`

Introduced before R2006a

mexEvalStringWithTrap (C and Fortran)

Execute MATLAB command in caller workspace and capture error information

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mexEvalStringWithTrap(command)
character*(*) command
```

Arguments

`command`

String containing the MATLAB command to execute

Returns

Object `ME` of class `MException`

Description

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

See Also

`mexEvalString`, `MException`, `mexCallMATLAB`

Introduced before R2006a

mexFunction (C and Fortran)

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

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer nlhs, nrhs
mwPointer plhs(*), prhs(*)
```

Arguments

`nlhs`

Number of expected output `mxArray`s

`plhs`

Array of pointers to the expected output `mxArray`s

`nrhs`

Number of input `mxArray`s

`prhs`

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

Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of the gateway function in C (subroutine in Fortran) which every MEX file requires. When you invoke a MEX function, MATLAB software finds and loads the corresponding MEX file of the same name. MATLAB then searches for a symbol named `mexFunction` within the MEX file. If it finds one, it calls the MEX function using the address of the `mexFunction` symbol. MATLAB displays an error message if it cannot find a routine named `mexFunction` inside the MEX file.

When you invoke a MEX file, MATLAB automatically seeds `nlhs`, `plhs`, `nrhs`, and `prhs` with the calling arguments. 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-side output arguments, and the `d,e,f...` are right-side input arguments. The arguments `nlhs` and `nrhs` contain the number of left side and right side arguments, respectively. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is an array whose length is `nlhs`, where your function must set pointers for the output `mxArray`s.

Note It is possible to return an output value even if `nlhs = 0`, which corresponds to returning the result in the `ans` variable.

To experiment with passing input arguments, build the `mexfunction.c` example, following the instructions in “Table of MEX File Source Code Files”.

Examples

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

- `mexfunction.c`
- `mexlockf.F`

See Also

Topics

"Introducing MEX Files"

Introduced before R2006a

mexFunctionName (C and Fortran)

Name of current MEX function

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
character*(*) mexFunctionName()
```

Returns

Name of the current MEX function.

Description

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

Examples

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

- `mexgetarray.c`

Introduced before R2006a

mexGet (C)

Value of specified graphics property

Note Do not use `mexGet`. Use `mxGetProperty` instead.

C Syntax

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

Arguments

`handle`

Handle to a particular graphics object

`property`

Graphics property

Returns

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

Description

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

See Also

`mxGetProperty`, `mxSetProperty`

mexGetVariable (C and Fortran)

Copy of variable from specified workspace

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mexGetVariable(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`

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

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

`varname`

Name of the variable to copy

Returns

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

Description

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

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

Examples

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

- `mexgetarray.c`

See Also

`mexGetVariablePtr`, `mexPutVariable`, `mxDestroyArray`

mexGetVariablePtr (C and Fortran)

Read-only pointer to variable from another workspace

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mexGetVariablePtr(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`

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

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

`varname`

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

Returns

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

Description

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

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

See Also

`mexGetVariable`

Introduced before R2006a

mexIsLocked (C and Fortran)

Determine if MEX-file is locked

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mexIsLocked()
```

Returns

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

Description

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

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

Examples

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

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

See Also

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

mexLock (C and Fortran)

Prevent clearing MEX-file from memory

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexLock()
```

Description

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

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

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

Examples

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

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

See Also

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

mexMakeArrayPersistent (C and Fortran)

Make array persist after MEX file completes

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexMakeArrayPersistent(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray` created by an `mxCreate*` function

Description

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

Do not assign an array created with the `mexMakeArrayPersistent` function to the `plhs` output argument of a MEX file.

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

See Also

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

Introduced before R2006a

mexMakeMemoryPersistent (C and Fortran)

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

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexMakeMemoryPersistent(ptr)
mwPointer ptr
```

Arguments

`ptr`

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

Description

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

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

See Also

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

Introduced before R2006a

mexPrintf (C and Fortran)

ANSI C PRINTF-style output routine

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mexPrintf(message)
character*(*) message
```

Arguments

`message`

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

`...`

In C, any arguments used in the message. Each argument must have a corresponding conversion specification. Refer to your C documentation for `printf` conversion tables.

Returns

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

Description

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

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

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

Examples

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

- `mexfunction.c`

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

- `phonebook.c`

See Also

`sprintf`, `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

mexPutVariable (C and Fortran)

Array from MEX-function into specified workspace

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mexPutVariable(workspace, varname, pm)
character*(*) workspace, varname
mwPointer pm
```

Arguments

`workspace`

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

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

`varname`

Name of `mxArray` in the workspace

`pm`

Pointer to the `mxArray`

Returns

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

Description

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

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

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

```
Peaches
1     2     3     4
```

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

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

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

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

Examples

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

- `mexgetarray.c`

See Also

mexGetVariable

mexSet (C)

Set value of specified graphics property

Note Do not use `mexSet`. Use `mxSetProperty` instead.

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 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`.

See Also

`mxGetProperty`, `mxSetProperty`

mexSetTrapFlag (C and Fortran)

Control response of MEXCALLMATLAB to errors

C Syntax

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

Note `mexSetTrapFlag` will be removed in a future version. Use `mexCallMATLABWithTrap` instead.

Fortran Syntax

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

Arguments

`trapflag`

Control flag. Possible values are:

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

Description

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

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

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

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

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

See Also

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

Introduced in R2008b

mexUnlock (C and Fortran)

Allow clearing MEX-file from memory

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexUnlock()
```

Description

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

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

Examples

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

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

See Also

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

mexWarnMsgIdAndTxt (C and Fortran)

Warning message with identifier

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mexWarnMsgIdAndTxt(warningid, warningmsg)
character*(*) warningid, warningmsg
```

Arguments

`warningid`

String containing a MATLAB message identifier. For information on creating identifiers, see “Message Identifiers”.

`warningmsg`

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

`...`

In C, any arguments used in the message. Each argument must have a corresponding conversion specification. Refer to your C documentation for `printf` conversion tables.

Description

The `mexWarnMsgIdAndTxt` function writes a warning message to the MATLAB command prompt. The warnings displayed are the same as warnings issued by the

MATLAB `warning` function. To control the information displayed or suppressed, call the `warning` function with the desired settings before calling your MEX-file.

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

See Also

`mexErrMsgIdAndTxt`, `warning`

Introduced before R2006a

mexWarnMsgTxt (C and Fortran)

Warning message

Note `mexWarnMsgTxt` is not recommended. Use `mexWarnMsgIdAndTxt` instead.

C Syntax

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

Fortran Syntax

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

Arguments

`warningmsg`

String containing the warning message to display

Description

`mexWarnMsgTxt` causes MATLAB software to display the contents of `warningmsg`. `mexWarnMsgTxt` does not terminate the MEX-file.

See Also

`mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

mwIndex (C and Fortran)

Type for index values

Description

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

The C header file containing this type is:

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

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

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

See Also

`mex`, `mwSize`, `mwSignedIndex`

Introduced before R2006a

mwPointer (Fortran)

Platform-independent pointer type

Description

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

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

The Fortran header file containing this type is:

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

Examples

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

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

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

Introduced in R2006a

mwSignedIndex (C and Fortran)

Signed integer type for size values

Description

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

The C header file containing this type is:

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

The Fortran header file containing this type is:

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

See Also

`mwSize`, `mwIndex`

Introduced in R2009a

mwSize (C and Fortran)

Type for size values

Description

`mwSize` is a type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. By default, `mwSize` is equivalent to `size_t` in C. `mwSize` is an unsigned type, meaning a nonnegative integer.

When using the `mex -compatibleArrayDims` switch, `mwSize` is equivalent to `int` in C. In Fortran, `mwSize` is similarly equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

The C header file containing this type is:

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

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

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

See Also

`mex`, `mwIndex`, `mwSignedIndex`

Introduced before R2006a

mxAddField (C and Fortran)

Add field to structure array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxAddField(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldname`

Name of the field you want to add

Returns

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

Description

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

See Also

`mxRemoveField`, `mxSetFieldByNumber`

Introduced before R2006a

mxArray (C)

Type for MATLAB array

Description

The fundamental type underlying MATLAB data. For information on how the MATLAB array works with MATLAB-supported variables, see “MATLAB Data”.

`mxArray` is a C language opaque type.

All C MEX-files start with a gateway routine, called `mexFunction`, which requires `mxArray` for both input and output parameters. For information about the C MEX-file gateway routine, see “Components of MEX File”.

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

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

The header file containing this type is:

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

Example

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

- `mxcreatecharmatrixfromstr.c`

See Also

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

Tips

- For information about data in MATLAB language scripts and functions, see “Data Types”.
- For troubleshooting `mxArray` errors in other MathWorks products, search the documentation for that product, or see MATLAB Answers™ topic "Subscripting into an mxArray is not supported".

Introduced before R2006a

mxArrayToString (C)

Array to string

C Syntax

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

Arguments

`array_ptr`

Pointer to `mxCHAR` array.

Returns

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

Description

Call `mxArrayToString` to copy the character data of an `mxCHAR` array into a C-style string. The C-style string is always terminated with a `NULL` character and stored in column-major order.

If the array contains multiple rows, the rows are copied column-wise into a single array.

This function is similar to `mxGetString`, except that:

- It does not require the length of the string as an input.
- It supports both multi-byte and single-byte encoded characters. On Windows and Linux® platforms, the default encoding is specified by the user locale setting.

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

Examples

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

- `mexatexit.c`

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

- `mxcreatecharmatrixfromstr.c`

See Also

`mxArrayToUTF8String`, `mxCreateCharArray`, `mxCreateCharMatrixFromStrings`, `mxCreateString`, `mxGetString`

Introduced before R2006a

mxArrayToUTF8String (C)

Array to string in UTF-8 encoding

C Syntax

```
#include "matrix.h"
char *mxArrayToUTF8String(const mxArray *array_ptr);
```

Arguments

`array_ptr`

Pointer to `mxCHAR` array.

Returns

C-style string in UTF-8 encoding. Returns `NULL` on failure. Possible reasons for failure include out of memory and specifying an array that is not an `mxCHAR` array.

Description

Call `mxArrayToUTF8String` to copy the character data of an `mxCHAR` array into a C-style string. The data is stored in column-major order. If the array contains multiple rows, the rows are copied column-wise into a single array.

`mxArrayToUTF8String` does not free the dynamic memory that the `char` pointer points to. Use `mxFree` to free memory.

See Also

`mxArrayToString`, `mxFree`, `mxCreateCharArray`, `mxCreateString`, `mxGetString`

Introduced in R2015a

mxAssert (C)

Check assertion value for debugging purposes

C Syntax

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

Arguments

`expr`

Value of assertion

`error_message`

Description of why assertion failed

Description

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

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

Assertions are a way of maintaining internal consistency of logic. Use them to keep yourself from misusing your own code and to prevent logical errors from propagating before they are caught. Do not use assertions to prevent users of your code from misusing it.

Assertions can be taken out of your code by the C preprocessor. You can use these checks during development and then remove them when the code works properly, letting you use them for troubleshooting during development without slowing down the final product.

See Also

`mxAssertS`, `mexErrMsgIdAndTxt`

Introduced before R2006a

mxAssertS (C)

Check assertion value without printing assertion text

C Syntax

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

Arguments

`expr`

Value of assertion

`error_message`

Description of why assertion failed

Description

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

See Also

`mxAssert`

Introduced before R2006a

mxCalcSingleSubscript (C and Fortran)

Offset from first element to desired element

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwIndex mxCalcSingleSubscript(pm, nsubs, subs)
mwPointer pm
mwSize nsubs
mwIndex subs
```

Arguments

`pm`

Pointer to an `mxArray`

`nsubs`

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

subs

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

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

Returns

The number of elements, or index, between the start of the `mxArray` and the specified subscript. This number is the linear index equivalent of the subscripts. Many Matrix Library routines (for example, `mxGetField`) require an index as an argument.

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

Description

Call `mxCalcSingleSubscript` to determine how many elements there are between the beginning of the `mxArray` and a given element of that `mxArray`. The function converts subscripts to linear indices.

For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the first element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have. For examples showing the internal representation, see “Data Storage”.

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

Examples

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

- `mxcalcsinglesubscript.c`

See Also

`mxGetCell`, `mxSetCell`

Introduced before R2006a

mxCalloc (C and Fortran)

Allocate dynamic memory for array, initialized to 0, using MATLAB memory manager

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCalloc(n, size)
mwSize n, size
```

Arguments

`n`

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

`size`

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

Returns

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

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

Description

`mxCalloc` allocates contiguous heap space sufficient to hold `n` elements of `size` bytes each, and initializes this newly allocated memory to `0`. Use `mxCalloc` instead of the ANSI C `calloc` function to allocate memory in MATLAB applications.

In MEX files, but not MAT or engine applications, `mxCalloc` registers the allocated memory with the MATLAB memory manager. When control returns to the MATLAB prompt, the memory manager then automatically frees, or deallocates, this memory.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using the `mxSetPr` function, MATLAB is responsible for freeing the memory.

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

If you do not assign this data to an output argument, and you want it to persist after the MEX file completes, call `mexMakeMemoryPersistent` after calling this function. If you write a MEX file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX file is cleared.

Examples

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

- `explore.c`

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

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

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

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

See Also

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

Introduced before R2006a

mxChar (C)

Type for string array

Description

MATLAB stores an `mxArray` string as type `mxChar` to represent the C-style `char` type. MATLAB uses 16-bit unsigned integer character encoding for Unicode characters.

The header file containing this type is:

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

Examples

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

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

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

- `explore.c`

See Also

`mxCreateCharArray`

Tips

- For information about data in MATLAB language scripts and functions, see “Data Types”.

Introduced before R2006a

mxClassID (C)

Enumerated value identifying class of array

C Syntax

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

Constants

`mxUNKNOWN_CLASS`

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

`mxCELL_CLASS`

Identifies a cell `mxArray`.

`mxSTRUCT_CLASS`

Identifies a structure `mxArray`.

`mxLOGICAL_CLASS`

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

`mxCHAR_CLASS`

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

`mxVOID_CLASS`

Reserved.

`mxDOUBLE_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxSINGLE_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxINT8_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxUINT8_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxINT16_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxUINT16_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxINT32_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

`mxUINT32_CLASS`

Identifies a numeric `mxArray` whose data is stored as the type specified in the MATLAB Primitive Types table.

mxINT64_CLASS

Identifies a numeric mxArray whose data is stored as the type specified in the MATLAB Primitive Types table.

mxUINT64_CLASS

Identifies a numeric mxArray whose data is stored as the type specified in the MATLAB Primitive Types table.

mxFUNCTION_CLASS

Identifies a function handle mxArray.

Description

Various Matrix Library functions require or return an mxClassID argument. mxClassID identifies how the mxArray represents its data elements.

The following table shows MATLAB types with their equivalent C types. Use the type from the right-most column for reading mxArrays with the mxClassID value shown in the left column.

MATLAB Primitive Types

mxClassID Value	MATLAB Type	MEX Type	C Primitive Type
mxINT8_CLASS	int8	int8_T	char, byte
mxUINT8_CLASS	uint8	uint8_T	unsigned char, byte
mxINT16_CLASS	int16	int16_T	short
mxUINT16_CLASS	uint16	uint16_T	unsigned short
mxINT32_CLASS	int32	int32_T	int
mxUINT32_CLASS	uint32	uint32_T	unsigned int
mxINT64_CLASS	int64	int64_T	long long
mxUINT64_CLASS	uint64	uint64_T	unsigned long long
mxSINGLE_CLASS	single	float	float
mxDOUBLE_CLASS	double	double	double

Examples

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

- `explore.c`

See Also

`mxGetClassID`, `mxCreateNumericArray`

Introduced before R2006a

mxClassIDFromClassName (Fortran)

Identifier corresponding to class

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxClassIDFromClassName(classname)
character*(*) classname
```

Arguments

`classname`

`character` array specifying a MATLAB class name. For a list of valid `classname` choices, see the `mxIsClass` reference page.

Returns

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

Description

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

Examples

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

- `matsqint8.F`

See Also

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

Introduced before R2006a

mxComplexity (C)

Flag specifying whether array 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 Matrix Library functions require an `mxComplexity` argument. You can set an `mxComplex` argument to either `mxREAL` or `mxCOMPLEX`.

Examples

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

- `mxcalcsinglesubscript.c`

See Also

`mxCreateNumericArray`, `mxCreateDoubleMatrix`, `mxCreateSparse`

Introduced before R2006a

mxCopyCharacterToPtr (Fortran)

CHARACTER values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyCharacterToPtr(y, px, n)
character*(*) y
mwPointer px
mwSize n
```

Arguments

`y`

  `character` Fortran array

`px`

  Pointer to `character` or `name` array

`n`

  Number of elements to copy

Description

`mxCopyCharacterToPtr` copies `n` `character` values from the Fortran `character` array `y` into the MATLAB character vector 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`

Introduced before R2006a

mxCopyComplex16ToPtr (Fortran)

COMPLEX*16 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyComplex16ToPtr(y, pr, pi, n)
complex*16 y(n)
mwPointer pr, pi
mwSize n
```

Arguments

`y`

  `COMPLEX*16` Fortran array

`pr`

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

`pi`

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

`n`

  Number of elements to copy

Description

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

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

Examples

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

- `convec.F`

See Also

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

Introduced before R2006a

mxCopyComplex8ToPtr (Fortran)

COMPLEX*8 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyComplex8ToPtr(y, pr, pi, n)
complex*8 y(n)
mwPointer pr, pi
mwSize n
```

Arguments

`y`

    `COMPLEX*8` Fortran array

`pr`

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

`pi`

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

`n`

    Number of elements to copy

Description

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

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

See Also

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

Introduced before R2006a

mxCopyInteger1ToPtr (Fortran)

INTEGER*1 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyInteger1ToPtr(y, px, n)
integer*1 y(n)
mwPointer px
mwSize n
```

Arguments

`y`

  `INTEGER*1` Fortran array

`px`

  Pointer to the real or imaginary data of the array

`n`

  Number of elements to copy

Description

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

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

Examples

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

- `matsqint8.F`

See Also

`mxCopyPtrToInteger1`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Introduced before R2006a

mxCopyInteger2ToPtr (Fortran)

INTEGER*2 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyInteger2ToPtr(y, px, n)
integer*2 y(n)
mwPointer px
mwSize n
```

Arguments

`y`

    `INTEGER*2` Fortran array

`px`

    Pointer to the real or imaginary data of the array

`n`

    Number of elements to copy

Description

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

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

See Also

`mxCopyPtrToInteger2`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Introduced before R2006a

mxCopyInteger4ToPtr (Fortran)

INTEGER*4 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyInteger4ToPtr(y, px, n)
integer*4 y(n)
mwPointer px
mwSize n
```

Arguments

`y`

    `INTEGER*4` Fortran array

`px`

    Pointer to the real or imaginary data of the array

`n`

    Number of elements to copy

Description

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

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

See Also

`mxCopyPtrToInteger4`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Introduced before R2006a

mxCopyPtrToCharacter (Fortran)

CHARACTER values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToCharacter(px, y, n)
mwPointer px
character*(*) y
mwSize n
```

Arguments

`px`

Pointer to `character` or `name` array

`y`

`character` Fortran array

`n`

Number of elements to copy

Description

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

Examples

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

- `matdemo2.F`

See Also

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

Introduced before R2006a

mxCopyPtrToComplex16 (Fortran)

COMPLEX*16 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToComplex16(pr, pi, y, n)
mwPointer pr, pi
complex*16 y(n)
mwSize n
```

Arguments

`pr`

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

`pi`

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

`y`

`COMPLEX*16` Fortran array

`n`

Number of elements to copy

Description

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

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

Examples

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

- convec.F

See Also

mxCopyComplex16ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

Introduced before R2006a

mxCopyPtrToComplex8 (Fortran)

COMPLEX*8 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToComplex8(pr, pi, y, n)
mwPointer pr, pi
complex*8 y(n)
mwSize n
```

Arguments

`pr`

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

`pi`

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

`y`

`COMPLEX*8` Fortran array

`n`

Number of elements to copy

Description

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

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

See Also

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

Introduced before R2006a

mxCopyPtrToInteger1 (Fortran)

INTEGER*1 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToInteger1(px, y, n)
mwPointer px
integer*1 y(n)
mwSize n
```

Arguments

`px`

Pointer to the real or imaginary data of the array

`y`

`INTEGER*1` Fortran array

`n`

Number of elements to copy

Description

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

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

Examples

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

- `matsqint8.F`

See Also

`mxCopyInteger1ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Introduced before R2006a

mxCopyPtrToInteger2 (Fortran)

INTEGER*2 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToInteger2(px, y, n)
mwPointer px
integer*2 y(n)
mwSize n
```

Arguments

`px`

Pointer to the real or imaginary data of the array

`y`

`INTEGER*2` Fortran array

`n`

Number of elements to copy

Description

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

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

See Also

`mxCopyInteger2ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Introduced before R2006a

mxCopyPtrToInteger4 (Fortran)

INTEGER*4 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToInteger4(px, y, n)
mwPointer px
integer*4 y(n)
mwSize n
```

Arguments

`px`

Pointer to the real or imaginary data of the array

`y`

`INTEGER*4` Fortran array

`n`

Number of elements to copy

Description

`mxCopyPtrToInteger4` copies `n` `INTEGER*4` values from the MATLAB array pointed to by `px`, either a real or an imaginary array, into the Fortran `INTEGER*4` array `y`.

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

See Also

`mxCopyInteger4ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Introduced before R2006a

mxCopyPtrToPtrArray (Fortran)

Pointer values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToPtrArray(px, y, n)
mwPointer px
mwPointer y(n)
mwSize n
```

Arguments

`px`

Pointer to pointer array

`y`

Fortran array of `mwPointer` values

`n`

Number of pointers to copy

Description

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

Examples

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

- `matdemo2.F`

See Also

`matGetDir`, `mxCopyPtrToCharacter`

Introduced before R2006a

mxCopyPtrToReal4 (Fortran)

REAL*4 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToReal4(px, y, n)
mwPointer px
real*4 y(n)
mwSize n
```

Arguments

`px`

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

`y`

`REAL*4` Fortran array

`n`

Number of elements to copy

Description

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

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

See Also

`mxCopyReal4ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Introduced before R2006a

mxCopyPtrToReal8 (Fortran)

REAL*8 values from pointer array to Fortran array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToReal8(px, y, n)
mwPointer px
real*8 y(n)
mwSize n
```

Arguments

`px`

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

`y`

`REAL*8` Fortran array

`n`

Number of elements to copy

Description

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

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

Examples

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

- `fengdemo.F`

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

- `timestwo.F`
- `xtimesy.F`

See Also

`mxCopyReal8ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Introduced before R2006a

mxCopyReal4ToPtr (Fortran)

REAL*4 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyReal4ToPtr(y, px, n)
real*4 y(n)
mwPointer px
mwSize n
```

Arguments

`y`

`REAL*4` Fortran array

`px`

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

`n`

Number of elements to copy

Description

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

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

See Also

`mxCopyPtrToReal4`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Introduced before R2006a

mxCopyReal8ToPtr (Fortran)

REAL*8 values from Fortran array to pointer array

Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyReal8ToPtr(y, px, n)
real*8 y(n)
mwPointer px
mwSize n
```

Arguments

`y`

  `REAL*8` Fortran array

`px`

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

`n`

  Number of elements to copy

Description

`mxCopyReal8ToPtr` copies `n` `REAL*8` values from the Fortran `REAL*8` array `y` into the MATLAB array pointed to by `px`, either a `pr` or `pi` array.

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

Examples

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

- `matdemo1.F`
- `fengdemo.F`

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

- `timestwo.F`
- `xtimesy.F`

See Also

`mxCopyPtrToReal8`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Introduced before R2006a

mxCreateCellArray (C and Fortran)

N-D cell array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCellArray(ndim, dims)
mwSize ndim
mwSize dims(ndim)
```

Arguments

`ndim`

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

`dims`

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

Returns

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

Description

Use `mxCreateCellArray` to create a cell `mxArray` with size defined by `ndim` and `dims`. For example, in C, to establish a three-dimensional cell `mxArray` having dimensions `4`-by-`8`-by-`7`, set:

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

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

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

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

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

Examples

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

- `phonebook.c`

See Also

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

Introduced before R2006a

mxCreateCellMatrix (C and Fortran)

2-D cell array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCellMatrix(m, n)
mwSize m, n
```

Arguments

`m`

Number of rows

`n`

Number of columns

Returns

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

Description

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

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

Examples

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

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

See Also

`mxCreateCellArray`

Introduced before R2006a

mxCreateCharArray (C and Fortran)

N-D `mxChar` array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCharArray(ndim, dims)
mwSize ndim
mwSize dims(ndim)
```

Arguments

`ndim`

Number of dimensions in the `mxArray`, specified as a positive number. If you specify 0, 1, or 2, `mxCreateCharArray` creates a two-dimensional `mxArray`.

`dims`

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

Returns

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

Description

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

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

See Also

`mxCreateCharMatrixFromStrings`, `mxCreateString`

mxCreateCharMatrixFromStrings (C and Fortran)

2-D `mxChar` array initialized to specified value

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCharMatrixFromStrings(m, str)
mwSize m
character*(*) str(m)
```

Arguments

`m`

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

`str`

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

Returns

Pointer to the created `mxArray`, if successful. If unsuccessful in a standalone (non-MEX file) application, returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`. Another possible reason for failure is that `str` contains fewer than `m` strings.

Description

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

The `mxArray` represents its data elements as `mxChar` rather than as C `char`.

Examples

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

- `mxcreatecharmatrixfromstr.c`

See Also

`mxCreateCharArray`, `mxCreateString`, `mxGetString`

Introduced before R2006a

mxCreateDoubleMatrix (C and Fortran)

2-D, double-precision, floating-point array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateDoubleMatrix(m, n, ComplexFlag)
mwSize m, n
integer*4 ComplexFlag
```

Arguments

`m`

Number of rows

`n`

Number of columns

`ComplexFlag`

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

Returns

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

Description

Use `mxCreateDoubleMatrix` to create an `m`-by-`n` `mxArray`. `mxCreateDoubleMatrix` initializes each element in the `pr` array to `0`. If you set `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran), `mxCreateDoubleMatrix` also initializes each element in the `pi` array to `0`.

If you set `ComplexFlag` to `mxREAL` in C (`0` in Fortran), `mxCreateDoubleMatrix` allocates enough memory to hold `m`-by-`n` real elements. If you set `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran), `mxCreateDoubleMatrix` allocates enough memory to hold `m`-by-`n` real elements and `m`-by-`n` imaginary elements.

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

Examples

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

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

For Fortran examples, see:

- `convec.F`
- `dblmat.F`
- `matsq.F`

- `timestwo.F`
- `xtimesy.F`

See Also

`mxCreateNumericArray`

Introduced before R2006a

mxCreateDoubleScalar (C and Fortran)

Scalar, double-precision array initialized to specified value

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateDoubleScalar(value)
real*8 value
```

Arguments

`value`

Value to which you want to initialize the array

Returns

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

Description

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

See Also

mxGetPr, mxCreateDoubleMatrix

Alternatives

C Language

In C, you can replace the statements:

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

with a call to mxCreateDoubleScalar:

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

Fortran Language

In Fortran, you can replace the statements:

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

with a call to mxCreateDoubleScalar:

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

Introduced before R2006a

mxCreateLogicalArray (C)

N-D logical array

C Syntax

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

Arguments

`ndim`

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

`dims`

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

Returns

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

Description

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

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

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

See Also

`mxCreateLogicalMatrix`, `mxCreateSparseLogicalMatrix`, `mxCreateLogicalScalar`

Introduced before R2006a

mxCreateLogicalMatrix (C)

2-D logical array

C Syntax

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

Arguments

`m`

Number of rows

`n`

Number of columns

Returns

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

Description

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

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

See Also

`mxCreateLogicalArray`, `mxCreateSparseLogicalMatrix`, `mxCreateLogicalScalar`

Introduced before R2006a

mxCreateLogicalScalar (C)

Scalar, logical array

C Syntax

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

Arguments

`value`

Logical value to which you want to initialize the array

Returns

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

Description

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

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

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

See Also

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

Introduced before R2006a

mxCreateNumericArray (C and Fortran)

N-D numeric array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateNumericArray(ndim, dims, classid, ComplexFlag)
mwSize ndim
mwSize dims(ndim)
integer*4 classid, ComplexFlag
```

Arguments

`ndim`

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

`dims`

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

`classid`

Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` in C causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer. In Fortran, use the function `mxClassIDFromClassName` to derive the `classid` value from a MATLAB class name. See the Description on page 1-403 section for more information.

`ComplexFlag`

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

Returns

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

Description

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

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

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

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

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

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

Examples

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

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

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

- mxisfinite.c

See Also

mxClassId, mxClassIdFromClassName, mxComplexity, mxDestroyArray, mxCreateUninitNumericArray, mxCreateNumericMatrix

Introduced before R2006a

mxCreateNumericMatrix (C and Fortran)

2-D numeric matrix

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateNumericMatrix(m, n, classid, ComplexFlag)
mwSize m, n
integer*4 classid, ComplexFlag
```

Arguments

`m`

Number of rows

`n`

Number of columns

`classid`

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

`ComplexFlag`

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

Returns

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

Description

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

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

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

Examples

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

- arrayFillGetPr.c

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

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

See Also

mxClassId, mxClassIdFromClassName, mxComplexity, mxDestroyArray, mxCreateUninitNumericMatrix, mxCreateNumericArray

Introduced before R2006a

mxCreateSparse (C and Fortran)

2-D sparse array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateSparse(m, n, nzmax, ComplexFlag)
mwSize m, n, nzmax
integer*4 ComplexFlag
```

Arguments

`m`

Number of rows

`n`

Number of columns

`nzmax`

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

`ComplexFlag`

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

Returns

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

Description

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

mxCreateSparse allocates space for:

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

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

Examples

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

- fulltosparse.c
- fulltosparse.F

See Also

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

Introduced before R2006a

mxCreateSparseLogicalMatrix (C)

2-D, sparse, logical array

C Syntax

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

Arguments

`m`

Number of rows

`n`

Number of columns

`nzmax`

Number of elements that `mxCreateSparseLogicalMatrix` should allocate to hold the data. Set the value of `nzmax` to be greater than or equal to the number of nonzero elements you plan to put into the `mxArray`, but make sure that `nzmax` is less than or equal to `m*n`. `nzmax` is greater than or equal to 1.

Returns

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

Description

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

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

See Also

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

Introduced before R2006a

mxCreateString (C and Fortran)

1-N array initialized to specified string

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateString(str)
character*(*) str
```

Arguments

`str`

String used to initialize `mxArray` data. Only ASCII characters are supported.

Returns

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

Description

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

`mxCreateString` supports both multi-byte and single-byte encoded characters. On Windows and Linux platforms, the default encoding is specified by the user locale setting.

Free the `mxArray` when you are finished using it, by calling `mxDestroyArray`.

Examples

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

- `revord.c`
- `revord.F`

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

- `mxcreatestructarray.c`
- `mxisclass.c`

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

- `matdemo1.F`

See Also

`mxCreateCharMatrixFromStrings`, `mxCreateCharArray`

Introduced before R2006a

mxCreateStructArray (C and Fortran)

N-D structure array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateStructArray(ndim, dims, nfields, fieldnames)
mwSize ndim
mwSize dims(ndim)
integer*4 nfields
character*(*) fieldnames(nfields)
```

Arguments

`ndim`

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

`dims`

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

`nfields`

Number of fields in each element. Positive integer.

`fieldnames`

List of field names. Field names must be valid MATLAB identifiers, which means they cannot be `NULL` or empty.

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

Returns

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

Description

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

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

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

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

Examples

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

- `mxcreatestructarray.c`

See Also

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

Introduced before R2006a

mxCreateStructMatrix (C and Fortran)

2-D structure array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateStructMatrix(m, n, nfields, fieldnames)
mwSize m, n
integer*4 nfields
character*(*) fieldnames(nfields)
```

Arguments

`m`

Number of rows; must be a positive integer.

`n`

Number of columns; must be a positive integer.

`nfields`

Number of fields in each element.

`fieldnames`

List of field names.

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

Returns

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

Description

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

C Examples

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

- `phonebook.c`

See Also

`mxCreateStructArray`, `namelengthmax`

Introduced before R2006a

mxCreateUninitNumericArray (C)

Uninitialized N-D numeric array

C Syntax

```
#include "matrix.h"
mxArray *mxCreateUninitNumericArray(size_t ndim, size_t *dims,
  mxClassID classid, mxComplexity ComplexFlag);
```

Arguments

`ndim`

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

`dims`

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

`classid`

Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer.

`ComplexFlag`

If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX`. Otherwise, set `ComplexFlag` to `mxREAL`.

Returns

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

Description

Call `mxCreateUninitNumericArray` to create an N-dimensional `mxArray` in which all data elements have the numeric data type specified by `classid`. Data elements are not initialized.

`mxCreateUninitNumericArray` allocates dynamic memory to store the created `mxArray`. Call `mxDestroyArray` to deallocate the memory.

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

MATLAB Class Name	C **`classid`** Value
`int8`	`mxINT8_CLASS`
`uint8`	`mxUINT8_CLASS`
`int16`	`mxINT16_CLASS`
`uint16`	`mxUINT16_CLASS`
`int32`	`mxINT32_CLASS`
`uint32`	`mxUINT32_CLASS`
`int64`	`mxINT64_CLASS`
`uint64`	`mxUINT64_CLASS`
`single`	`mxSINGLE_CLASS`
`double`	`mxDOUBLE_CLASS`

See Also

`mxDestroyArray`, `mxCreateUninitNumericMatrix`, `mxCreateNumericArray`

Introduced in R2015a

mxCreateUninitNumericMatrix (C)

Uninitialized 2-D numeric matrix

C Syntax

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

Arguments

`m`

Number of rows

`n`

Number of columns

`classid`

Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer.

`ComplexFlag`

If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX`. Otherwise, set `ComplexFlag` to `mxREAL`.

Returns

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

Example

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

- mxcreateuninitnumericmatrix.c

Description

Call mxCreateUninitNumericMatrix to create a 2-D mxArray in which all data elements have the numeric data type specified by classid. Data elements are not initialized.

mxCreateUninitNumericMatrix allocates dynamic memory to store the created mxArray. Call mxDestroyArray to deallocate the memory.

The following table shows the C classid values that are equivalent to MATLAB classes.

MATLAB Class Name	C **classid** Value
int8	mxINT8_CLASS
uint8	mxUINT8_CLASS
int16	mxINT16_CLASS
uint16	mxUINT16_CLASS
int32	mxINT32_CLASS
uint32	mxUINT32_CLASS
int64	mxINT64_CLASS
uint64	mxUINT64_CLASS
single	mxSINGLE_CLASS
double	mxDOUBLE_CLASS

See Also

mxDestroyArray, mxCreateUninitNumericArray, mxCreateNumericMatrix

Introduced in R2015a

mxDestroyArray (C and Fortran)

Free dynamic memory allocated by MXCREATE* functions

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mxDestroyArray(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to the `mxArray` to free. If `pm` is a `NULL` pointer, the function does nothing.

Description

`mxDestroyArray` deallocates the memory occupied by the specified `mxArray` including:

- Characteristics fields of the `mxArray`, such as size (`m` and `n`) and type.
- Associated data arrays, such as `pr` and `pi` for complex arrays, and `ir` and `jc` for sparse arrays.
- Fields of structure arrays.
- Cells of cell arrays.

Do not call `mxDestroyArray` on an `mxArray`:

- returned in a left-side argument of a MEX file.

- returned by the `mxGetField` or `mxGetFieldByNumber` functions.
- returned by the `mxGetCell` function.

Examples

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

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

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

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

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

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

See Also

`mxCalloc`, `mxMalloc`, `mxFree`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

Introduced before R2006a

mxDuplicateArray (C and Fortran)

Make deep copy of array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxDuplicateArray(in)
mwPointer in
```

Arguments

`in`

Pointer to the `mxArray` you want to copy

Returns

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

Description

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

Examples

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

- phonebook.c

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

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

Introduced before R2006a

mxFree (C and Fortran)

Free dynamic memory allocated by mxCalloc, mxMalloc, mxRealloc, mxArrayToString, or mxArrayToUTF8String functions

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
subroutine mxFree(ptr)
mwPointer ptr
```

Arguments

`ptr`

Pointer to the beginning of any memory parcel allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`. If `ptr` is a `NULL` pointer, the function does nothing.

Description

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

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

In MEX files, but excluding MAT or engine standalone applications, the MATLAB memory management facility maintains a list of all memory allocated by the following functions:

- `mxCalloc`
- `mxMalloc`
- `mxRealloc`
- `mxArrayToString`
- `mxArrayToUTF8String`

The memory management facility automatically deallocates all parcels managed by a MEX file when the MEX file completes and control returns to the MATLAB prompt. `mxFree` also removes the memory parcel from the memory management list of parcels.

When `mxFree` appears in a MAT or engine standalone MATLAB application, it simply deallocates the contiguous heap space that begins at address `ptr`.

In MEX files, your use of `mxFree` depends on whether the specified memory parcel is persistent or nonpersistent. By default, memory parcels created by `mxCalloc`, `mxMalloc`, `mxRealloc`, `mxArrayToString`, and `mxArrayToUTF8String` are nonpersistent. The memory management facility automatically frees all nonpersistent memory whenever a MEX file completes. Thus, even if you do not call `mxFree`, MATLAB takes care of freeing the memory for you. Nevertheless, it is good programming practice to deallocate memory when you are through using it. Doing so generally makes the entire system run more efficiently.

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

Do not use `mxFree` for an `mxArray` created by any other functions in the Matrix Library API. Use `mxDestroyArray` instead.

Examples

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

- `mxcalcsinglesubscript.c`
- `mxcreatecharmatrixfromstr.c`

- `mxisfinite.c`
- `mxmalloc.c`
- `mxsetdimensions.c`

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

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

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

- `explore.c`

See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxMalloc`, `mxRealloc`, `mxArrayToString`, `mxArrayToUTF8String`

Introduced before R2006a

mxGetCell (C and Fortran)

Pointer to element in cell array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetCell(pm, index)
mwPointer pm
mwIndex index
```

Arguments

`pm`

Pointer to a cell `mxArray`

`index`

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

Returns

Pointer to the `i`th cell `mxArray` if successful. Otherwise, returns `NULL` in C (`0` in Fortran). Causes of failure include:

- Specifying the index of a cell array element that has not been populated.
- Specifying a `pm` that does not point to a cell `mxArray`.

- Specifying an `index` to an element outside the bounds of the `mxArray`.
- Insufficient heap space.

Do not call `mxDestroyArray` on an `mxArray` returned by the `mxGetCell` function.

Description

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

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

Examples

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

- `explore.c`

See Also

`mxCreateCellArray`, `mxIsCell`, `mxSetCell`

Introduced before R2006a

mxGetChars (C)

Pointer to character array data

C Syntax

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

Arguments

`array_ptr`

Pointer to an `mxArray`

Returns

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

Description

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

See Also

`mxGetString`

Introduced before R2006a

mxGetClassID (C and Fortran)

Class of array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetClassID(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Numeric identifier of the class (category) of the `mxArray` that `pm` points to. For a list of C-language class identifiers, see the `mxClassID` reference page. For user-defined types, `mxGetClassId` returns a unique value identifying the class of the array contents. Use `mxIsClass` to determine whether an array is of a specific user-defined type.

Description

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

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

Examples

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

- `explore.c`

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

- `phonebook.c`

See Also

`mxClassID`, `mxGetClassName`, `mxIsClass`

Introduced before R2006a

mxGetClassName (C and Fortran)

Class of array as string

Note Use `mxGetClassName` for classes defined without a `classdef` statement.

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
character*(*) mxGetClassName(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

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

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

Examples

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

- mexfunction.c

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

- mxisclass.c

See Also

mxGetClassID, mxIsClass

Introduced before R2006a

mxGetData (C and Fortran)

Pointer to real numeric data elements in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetData(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

In C, `mxGetData` returns a void pointer (`void *`). Since void pointers point to a value that has no type, cast the return value to the pointer type that matches the type specified by `pm`. To see how MATLAB types map to their equivalent C types, see the table on the `mxClassID` reference page.

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

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

Examples

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

- `explore.c`

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

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

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

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

See Also

`mxGetImagData`, `mxGetPr`, `mxClassID`

Introduced before R2006a

mxGetDimensions (C and Fortran)

Pointer to dimensions array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetDimensions(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`.

Returns

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

Description

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

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

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

Examples

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

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

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

- findnz.c
- phonebook.c

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

- explore.c

See Also

mxGetNumberOfDimensions

Introduced before R2006a

mxGetElementSize (C and Fortran)

Number of bytes required to store each data element

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetElementSize(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

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

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

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

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

- `doubleelement.c`
- `phonebook.c`

See Also

`mxGetM`, `mxGetN`

Introduced before R2006a

mxGetEps (C and Fortran)

Value of EPS

C Syntax

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

Fortran Syntax

```
real*8 mxGetEps
```

Returns

Value of the MATLAB `eps` variable

Description

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

Examples

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

- `mxgeteps.c`
- `mxgetepsf.F`

See Also

`mxGetInf`, `mxGetNan`

Introduced before R2006a

mxGetField (C and Fortran)

Pointer to field value from structure array, given index and field name

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetField(pm, index, fieldname)
mwPointer pm
mwIndex index
character*(*) fieldname
```

Arguments

`pm`

Pointer to a structure `mxArray`

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`fieldname`

Name of the field whose value you want to extract.

Returns

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

- Specifying an array pointer pm that does not point to a structure mxArray. To determine whether pm points to a structure mxArray, call mxIsStruct.
- Specifying an index to an element outside the bounds of the mxArray. For example, given a structure mxArray that contains 10 elements, you cannot specify an index greater than 9 in C (10 in Fortran).
- Specifying a nonexistent fieldname. Call mxGetFieldNameByNumber or mxGetFieldNumber to get existing field names.
- Insufficient heap space.

Description

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

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

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

Do not call mxDestroyArray on an mxArray returned by the mxGetField function.

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

In C, calling:

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

is equivalent to calling:

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

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

In Fortran, calling:

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

is equivalent to calling:

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

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

Examples

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

- `matreadstructarray.c`

See Also

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

Introduced before R2006a

mxGetFieldByNumber (C and Fortran)

Pointer to field value from structure array, given index and field number

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetFieldByNumber(pm, index, fieldnumber)
mwPointer pm
mwIndex index
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for more details on calculating an `index`.

`fieldnumber`

Position of the field whose value you want to extract

In C, the first field within each element has a field number of `0`, the second field has a field number of `1`, and so on. The last field has a field number of `N-1`, where `N` is the number of fields.

In Fortran, the first field within each element has a field number of `1`, the second field has a field number of `2`, and so on. The last field has a field number of `N`, where `N` is the number of fields.

Returns

Pointer to the `mxArray` in the specified field for the desired element, on success. Returns `NULL` in C (`0` in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include:

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

Description

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

Do not call `mxDestroyArray` on an `mxArray` returned by the `mxGetFieldByNumber` function.

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

In C, calling:

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

is equivalent to calling:

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

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

In Fortran, calling:

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

is equivalent to calling:

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

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

Examples

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

- `phonebook.c`

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

- `mxisclass.c`

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

- `explore.c`

See Also

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

Introduced before R2006a

mxGetFieldNameByNumber (C and Fortran)

Pointer to field name from structure array, given field number

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
character*(*) mxGetFieldNameByNumber(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

Position of the desired field. For instance, in C, to get the name of the first field, set `fieldnumber` to 0; to get the name of the second field, set `fieldnumber` to 1; and so on. In Fortran, to get the name of the first field, set `fieldnumber` to `1`; to get the name of the second field, set `fieldnumber` to `2`; and so on.

Returns

Pointer to the `n`th field name, on success. Returns `NULL` in C (`0` in Fortran) on failure. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

- Specifying a value of `fieldnumber` outside the bounds of the number of fields in the structure `mxArray`. In C, `fieldnumber` 0 represents the first field, and `fieldnumber` `N-1` represents the last field, where `N` is the number of fields in the structure `mxArray`. In Fortran, `fieldnumber` `1` represents the first field, and `fieldnumber` `N` represents the last field.

Description

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

Consider a MATLAB structure initialized to:

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

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

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

Examples

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

- `phonebook.c`

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

- `mxisclass.c`

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

- `explore.c`

See Also

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

Introduced before R2006a

mxGetFieldNumber (C and Fortran)

Field number from structure array, given field name

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetFieldNumber(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldname`

Name of a field in the structure `mxArray`

Returns

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

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

- Specifying the `fieldname` of a nonexistent field.

Description

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

For example, consider a MATLAB structure initialized to:

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

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

Calling:

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

is equivalent to calling:

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

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

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

Calling:

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

is equivalent to calling:

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

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

Examples

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

- `mxcreatestructarray.c`

See Also

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

Introduced before R2006a

mxGetImagData (C and Fortran)

Pointer to imaginary data elements in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetImagData(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

This function is like `mxGetPi`, except that in C it returns a `void *`. For more information, see the description for the `mxGetData` function.

Examples

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

- `explore.c`

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

- `mxisfinite.c`

See Also

`mxGetData`, `mxGetPi`

Introduced before R2006a

mxGetInf (C and Fortran)

Value of infinity

C Syntax

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

Fortran Syntax

```
real*8 mxGetInf
```

Returns

Value of infinity on your system.

Description

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

Operations that return infinity include:

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

Examples

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

- `mxgetinf.c`

See Also

`mxGetEps`, `mxGetNaN`

Introduced before R2006a

mxGetIr (C and Fortran)

Sparse matrix IR array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetIr(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a sparse `mxArray`

Returns

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

- Specifying a full (nonsparse) `mxArray`.
- Specifying a value for `pm` that is `NULL` in C (`0` in Fortran). This failure 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 `ir` is `nzmax`, the storage allocated for the sparse array, or `nnz`, the number of nonzero matrix elements. For example, if `nzmax` equals `100`, the `ir` array contains 100 integers.

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

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

Examples

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

- `fulltosparse.c`
- `fulltosparse.F`

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

- `mxsetdimensions.c`
- `mxsetnzmax.c`

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

- `explore.c`

See Also

`mxGetJc`, `mxGetNzmax`, `mxSetIr`, `mxSetJc`, `mxSetNzmax`, `nzmax`, `nnz`

Introduced before R2006a

mxGetJc (C and Fortran)

Sparse matrix JC array

C Syntax

```
#include "matrix.h"
mwIndex *mxGetJc(const mxArray *pm);
```

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetJc(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a sparse `mxArray`

Returns

Pointer to the first element in the `jc` array, if successful, and `NULL` in C (`0` in Fortran) otherwise. Possible causes of failure include

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

Description

Use `mxGetJc` to obtain the starting address of the `jc` array. The `jc` array is an integer array having `n+1` elements, where `n` is the number of columns in the sparse `mxArray`.

The values in the `jc` array indirectly indicate columns containing nonzero elements. For a detailed explanation of the `jc` array, see `mxSetJc`.

Examples

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

- `fulltosparse.c`
- `fulltosparse.F`

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

- `mxgetnzmax.c`
- `mxsetdimensions.c`
- `mxsetnzmax.c`

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

- `explore.c`

See Also

`mxGetIr`, `mxGetNzmax`, `mxSetIr`, `mxSetJc`, `mxSetNzmax`

Introduced before R2006a

mxGetLogicals (C)

Pointer to logical array data

C Syntax

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

Arguments

`array_ptr`

Pointer to an `mxArray`

Returns

Pointer to the first logical element in the `mxArray`. The result is unspecified if the `mxArray` is not a logical array.

Description

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

See Also

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

Introduced before R2006a

mxGetM (C and Fortran)

Number of rows in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetM(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Number of rows in the `mxArray` to which `pm` points.

Description

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

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

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

- `convec.c`
- `fulltosparse.c`
- `matrixDivide.c`
- `matrixDivideComplex.c`
- `revord.c`
- `timestwo.c`
- `xtimesy.c`

For Fortran examples, see:

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

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

- `mxmalloc.c`
- `mxsetdimensions.c`
- `mxgetnzmax.c`
- `mxsetnzmax.c`

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

- `explore.c`

- `mexlock.c`
- `yprime.c`

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

- `matdemo2.F`

See Also

`mxGetN`, `mxSetM`, `mxSetN`

Introduced before R2006a

mxGetN (C and Fortran)

Number of columns in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetN(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Number of columns in the `mxArray`.

Description

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

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

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

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

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

- `convec.c`
- `fulltosparse.c`
- `revord.c`
- `timestwo.c`
- `xtimesy.c`

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

- `mxmalloc.c`
- `mxsetdimensions.c`
- `mxgetnzmax.c`
- `mxsetnzmax.c`

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

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

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

- `matdemo2.F`

See Also

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

Introduced before R2006a

mxGetNaN (C and Fortran)

Value of NaN (Not-a-Number)

C Syntax

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

Fortran Syntax

```
real*8 mxGetNaN
```

Returns

Value of `NaN` (Not-a-Number) on your system

Description

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

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

Your system specifies the value of Not-a-Number. You cannot modify it.

C Examples

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

- `mxgetinf.c`

See Also

`mxGetEps`, `mxGetInf`

Introduced before R2006a

mxGetNumberOfDimensions (C and Fortran)

Number of dimensions in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwSize mxGetNumberOfDimensions(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

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

Examples

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

- `explore.c`

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

- `findnz.c`
- `fulltosparse.c`
- `phonebook.c`

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

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

See Also

`mxSetM`, `mxSetN`, `mxGetDimensions`

Introduced before R2006a

mxGetNumberOfElements (C and Fortran)

Number of elements in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetNumberOfElements(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Number of elements in the specified `mxArray`

Description

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

Note Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

Examples

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

- `findnz.c`
- `phonebook.c`

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

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxgetepsf.F`
- `mxgetinf.c`
- `mxisfinite.c`
- `mxsetdimensions.c`
- `mxsetdimensionsf.F`

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

- `explore.c`

See Also

`mxGetDimensions`, `mxGetM`, `mxGetN`, `mxGetClassID`, `mxGetClassName`

Introduced before R2006a

mxGetNumberOfFields (C and Fortran)

Number of fields in structure array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetNumberOfFields(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a structure `mxArray`

Returns

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

Description

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

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

Examples

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

- phonebook.c

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

- mxisclass.c

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

- explore.c

See Also

mxGetField, mxIsStruct, mxSetField

Introduced before R2006a

mxGetNzmax (C and Fortran)

Number of elements in IR, PR, and PI arrays

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwSize mxGetNzmax(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a sparse `mxArray`

Returns

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

Description

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

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

Examples

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

- `mxgetnzmax.c`
- `mxsetnzmax.c`

See Also

`mxSetNzmax`

Introduced before R2006a

mxGetPi (C and Fortran)

Imaginary data elements in array of type DOUBLE

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetPi(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray` of type `double`

Returns

Pointer to the imaginary data elements of the specified `mxArray`, on success. Returns `NULL` in C (`0` in Fortran) if there is no imaginary data or if there is an error.

Description

Use `mxGetPi` on arrays of type `double` only. Use `mxIsDouble` to validate the `mxArray` type. For other `mxArray` types, use `mxGetImagData`.

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

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

If any of the input matrices to a function are complex, MATLAB allocates the imaginary parts of all input matrices.

Examples

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

- `convec.c`
- `findnz.c`
- `fulltosparse.c`

For Fortran examples, see:

- `convec.F`

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

- `mxcalcsinglesubscript.c`
- `mxgetinf.c`
- `mxisfinite.c`
- `mxsetnzmax.c`

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

- `explore.c`
- `mexcallmatlab.c`

See Also

`mxGetPr`, `mxSetPi`, `mxSetPr`, `mxGetImagData`, `mxIsDouble`

Introduced before R2006a

mxGetPr (C and Fortran)

Real data elements in array of type DOUBLE

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetPr(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray` of type `double`

Returns

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

Description

Use `mxGetPr` on arrays of type `double` only. Use `mxIsDouble` to validate the `mxArray` type. For other `mxArray` types, use `mxGetData`.

Call `mxGetPr` to access the real data in the `mxArray` that `pm` points to. Once you have the starting address, you can access any other element in the `mxArray`.

Examples

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

- arrayFillGetPrDynamicData.c
- arrayFillGetPr.c
- convec.c
- doubleelement.c
- findnz.c
- fulltosparse.c
- matrixDivide.c
- matrixMultiply.c
- sincall.c
- timestwo.c
- timestwoalt.c
- xtimesy.c

For Fortran examples, see:

- convec.F
- dblmat.F
- fulltosparse.F
- matsq.F
- sincall.F
- timestwo.F
- xtimesy.F

See Also

mxGetPi, mxSetPi, mxSetPr, mxGetData, mxIsDouble

Introduced before R2006a

mxGetProperty (C and Fortran)

Value of public property of MATLAB object

C Syntax

```
#include "matrix.h"
mxArray *mxGetProperty(const mxArray *pa, mwIndex index,
         const char *propname);
```

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetProperty(pa, index, propname)
mwPointer pa
mwIndex index
character*(*) propname
```

Arguments

`pa`

Pointer to an `mxArray` which is an object.

`index`

Index of the desired element of the object array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`propname`

Name of the property whose value you want to extract.

Returns

Pointer to the mxArray of the specified propname on success. Returns NULL in C (0 in Fortran) if unsuccessful. Common causes of failure include:

- Specifying a nonexistent propname.
- Specifying a nonpublic propname.
- Specifying an index to an element outside the bounds of the mxArray. To test the index value, use mxGetNumberOfElements or mxGetM and mxGetN.
- Insufficient heap space.

Description

Call mxGetProperty to get the value held in the specified element. In pseudo-C terminology, mxGetProperty returns the value at:

```
pa[index].propname
```

mxGetProperty makes a copy of the value. If the property uses a large amount of memory, creating a copy might be a concern. There must be sufficient memory (in the heap) to hold the copy of the value.

Examples

Display Name Property of timeseries Object

Create a MEX file, dispproperty.c, in a folder on your MATLAB path.

```
/*=================================================================
 * dispproperty.c - Display timeseries Name property
 * This is a MEX file for MATLAB.
 * Copyright 2013 The MathWorks, Inc.
 * All rights reserved.
 *=================================================================*/

#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
```

```
{
  /* Check for proper number of arguments. */
  if(nrhs!=1) {
    mexErrMsgIdAndTxt( "MATLAB:dispproperty:invalidNumInputs",
            "One input required.");
  } else if(nlhs>1) {
    mexErrMsgIdAndTxt( "MATLAB:dispproperty:maxlhs",
            "Too many output arguments.");
  }
  /* Check for timeseries object. */
  if (!mxIsClass(prhs[0], "timeseries")) {
    mexErrMsgIdAndTxt( "MATLAB:dispproperty:invalidClass",
            "Input must be timeseries object.");
  }
  plhs[0] = mxGetProperty(prhs[0],0,"Name");

}
```

Build the MEX file.

```
mex('-v','dispproperty.c')
```

Create a `timeseries` object.

```
ts = timeseries(rand(5, 4),'Name','LaunchData');
```

Display name.

```
tsname = dispproperty(ts)
```

```
tsname =
LaunchData
```

Change Object Color

Open and build the `mexgetproperty.c` MEX file in the *`matlabroot`*`/extern/examples/mex` folder.

Limitations

- `mxGetProperty` is not supported for standalone applications, such as applications built with the MATLAB engine API.

See Also

`mxSetProperty`, `mxGetNumberOfElements`, `mxGetM`, `mxGetN`

Introduced in R2008a

mxGetScalar (C and Fortran)

Real component of first data element in array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
real*8 mxGetScalar(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`; cannot be a cell `mxArray`, a structure `mxArray`, or an empty `mxArray`.

Returns

The value of the first real (nonimaginary) element of the `mxArray`.

In C, `mxGetScalar` returns a `double`. If real elements in the `mxArray` are of a type other than `double`, then `mxGetScalar` automatically converts the scalar value into a `double`. To preserve the original data representation of the scalar, cast the return value to the desired data type.

If `pm` points to a sparse `mxArray`, then `mxGetScalar` returns the value of the first nonzero real element in the `mxArray`. If there are no nonzero elements, then the function returns `0`.

Description

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

Usually 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, then the function returns the value of the first real element. For example, if `pm` points to a two-dimensional `mxArray`, then `mxGetScalar` returns the value of the `(1,1)` element. If `pm` points to a three-dimensional `mxArray`, then the function returns the value of the `(1,1,1)` element; and so on.

Use `mxGetScalar` on a nonempty `mxArray` of type numeric, logical, or char only. To test for these conditions, use Matrix Library functions such as `mxIsEmpty`, `mxIsLogical`, `mxIsNumeric`, or `mxIsChar`.

If the input value to `mxGetScalar` is type `int64` or `uint64`, then the value might lose precision if it is greater than `flintmax`.

Examples

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

- `timestwoalt.c`
- `xtimesy.c`

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

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

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

- `mxsetdimensions.c`

See Also

`mxGetM`, `mxGetN`, `mxIsScalar`

mxGetString (C and Fortran)

mxChar array to C-style string or Fortran character array

C Syntax

```
#include "matrix.h"
int mxGetString(const mxArray *pm, char *str, mwSize strlen);
```

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetString(pm, str, strlen)
mwPointer pm
character*(*) str
mwSize strlen
```

Arguments

pm

Pointer to an mxChar array.

str

Starting location. mxGetString writes the character data into str and then, in C, terminates the string with a NULL character (in the manner of C strings). str can point to either dynamic or static memory.

strlen

Size in bytes of destination buffer pointed to by str. Typically, in C, you set strlen to 1 plus the number of elements in the mxArray to which pm points. To get the number of elements, use mxGetM or mxGetN.

Do not use with “Multibyte Encoded Characters” on page 1-497.

Returns

`0` on success or if `strlen` == `0`, and `1` on failure. Possible reasons for failure include:

- `mxArray` is not an `mxChar` array.
- `strlen` is not large enough to store the entire `mxArray`. If so, the function returns `1` and truncates the string.

Description

Call `mxGetString` to copy the character data of an `mxArray` into a C-style string in C or a `character` array in Fortran. The copied data starts at `str` and contains no more than `strlen-1` characters in C (no more than `strlen` characters in Fortran). In C, the C-style string is always terminated with a `NULL` character.

If the array contains multiple rows, the function copies them into a single array, one column at a time.

Multibyte Encoded Characters

Use this function only with characters represented in single-byte encoding schemes. For characters represented in multibyte encoding schemes, use the C function `mxArrayToString`. Fortran users must allocate sufficient space for the return string to avoid possible truncation.

Examples

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

- `mxmalloc.c`

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

- `explore.c`

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

- `revord.F`

See Also

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

Introduced before R2006a

mxIsCell (C and Fortran)

Determine whether input is cell array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsCell(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to an array having the class `mxCELL_CLASS`, and logical `0` (`false`) otherwise.

Description

Use `mxIsCell` to determine whether the specified array is a cell array.

In C, calling `mxIsCell` is equivalent to calling:

```
mxGetClassID(pm) == mxCELL_CLASS
```

In Fortran, calling `mxIsCell` is equivalent to calling:

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

Note `mxIsCell` does not answer the question “Is this `mxArray` a cell of a cell array?” An individual cell of a cell array can be of any type.

See Also

`mxIsClass`

Introduced before R2006a

mxIsChar (C and Fortran)

Determine whether input is `mxChar` array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsChar(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to an array having the class `mxCHAR_CLASS`, and logical `0` (`false`) otherwise.

Description

Use `mxIsChar` to determine whether `pm` points to an `mxChar` array.

In C, calling `mxIsChar` is equivalent to calling:

```
mxGetClassID(pm) == mxCHAR_CLASS
```

In Fortran, calling `mxIsChar` is equivalent to calling:

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

Examples

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

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

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

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

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

- `matdemo1.F`

See Also

`mxIsClass`, `mxGetClassID`

Introduced before R2006a

mxIsClass (C and Fortran)

Determine whether array is object of specified class

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsClass(pm, classname)
mwPointer pm
character*(*) classname
```

Arguments

`pm`

Pointer to an `mxArray`

`classname`

Array category to test. Specify `classname` as a string (not as an integer identifier). You can specify any one of the following predefined constants:

Value of classname	Corresponding Class
cell	mxCELL_CLASS
char	mxCHAR_CLASS
double	mxDOUBLE_CLASS
function_handle	mxFUNCTION_CLASS
int8	mxINT8_CLASS
int16	mxINT16_CLASS

Value of classname	Corresponding Class
`int32`	`mxINT32_CLASS`
`int64`	`mxINT64_CLASS`
`logical`	`mxLOGICAL_CLASS`
`single`	`mxSINGLE_CLASS`
`struct`	`mxSTRUCT_CLASS`
`uint8`	`mxUINT8_CLASS`
`uint16`	`mxUINT16_CLASS`
`uint32`	`mxUINT32_CLASS`
`uint64`	`mxUINT64_CLASS`
<class_name>	*<class_id>*
`unknown`	`mxUNKNOWN_CLASS`

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

Returns

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

Description

Each `mxArray` is tagged as being a certain type. Call `mxIsClass` to determine whether the specified `mxArray` has this type. MATLAB does not check if the class is derived from a base class.

In C:

```
mxIsClass(pm, "double");
```

is equivalent to calling either of these forms:

```
mxIsDouble(pm);
```

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

In Fortran:

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

is equivalent to calling either one of the following:

```
mxIsDouble(pm)
```

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

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

Examples

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

- `mxisclass.c`

See Also

`mxClassID`, `mxGetClassID`, `mxIsEmpty`, `mxGetClassName`

Introduced before R2006a

mxIsComplex (C and Fortran)

Determine whether data is complex

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsComplex(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` is a numeric array containing complex data, and logical `0` (`false`) otherwise. If `pm` points to a cell array or a structure array, `mxIsComplex` returns `false`.

Description

Use `mxIsComplex` to determine whether an imaginary part is allocated for an `mxArray`. If an `mxArray` is purely real and does not have any imaginary data, the imaginary pointer `pi` is `NULL` in C (`0` in Fortran). If an `mxArray` is complex, `pi` points to an array of numbers.

Examples

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

- mxisfinite.c
- mxgetinf.c

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

- convec.c
- convec.F
- fulltosparse.F
- phonebook.c

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

- explore.c
- yprime.c
- mexlock.c

See Also

mxIsNumeric

Introduced before R2006a

mxIsDouble (C and Fortran)

Determine whether mxArray represents data as double-precision, floating-point numbers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsDouble(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` stores its data as double-precision, floating-point numbers, and logical `0` (`false`) otherwise.

Description

Call `mxIsDouble` to determine whether the specified `mxArray` represents its real and imaginary data as double-precision, floating-point numbers.

Older versions of MATLAB software store all `mxArray` data as double-precision, floating-point numbers. However, starting with MATLAB Version 5 software, MATLAB can store real and imaginary data in various numerical formats.

In C, calling `mxIsDouble` is equivalent to calling:

```
mxGetClassID(pm) == mxDOUBLE_CLASS
```

In Fortran, calling `mxIsDouble` is equivalent to calling:

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

Examples

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

- `fulltosparse.c`
- `fulltosparse.F`

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

- `mxgeteps.c`
- `mxgetepsf.F`

See Also

`mxIsClass`, `mxGetClassID`

Introduced before R2006a

mxIsEmpty (C and Fortran)

Determine whether array is empty

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsEmpty(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` is empty, and logical `0` (`false`) otherwise.

Description

Use `mxIsEmpty` to determine whether an `mxArray` contains no data. An `mxArray` is empty if the size of any of its dimensions is 0.

Examples

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

- `mxisfinite.c`

See Also

`mxIsClass`

Introduced before R2006a

mxIsFinite (C and Fortran)

Determine whether input is finite

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsFinite(value)
real*8 value
```

Arguments

`value`

Double-precision, floating-point number to test

Returns

Logical `1` (`true`) if `value` is finite, and logical `0` (`false`) otherwise.

Description

Call `mxIsFinite` to determine whether `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

Examples

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

- `mxisfinite.c`

See Also

`mxIsInf`, `mxIsNan`

Introduced before R2006a

mxIsFromGlobalWS (C and Fortran)

Determine whether array was copied from MATLAB global workspace

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsFromGlobalWS(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the array was copied out of the global workspace, and logical `0` (`false`) otherwise.

Description

`mxIsFromGlobalWS` is useful for standalone MAT-file programs.

Examples

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

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

Introduced before R2006a

mxIsInf (C and Fortran)

Determine whether input is infinite

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInf(value)
real*8 value
```

Arguments

`value`

Double-precision, floating-point number to test

Returns

Logical `1` (`true`) if value is infinite, and logical `0` (`false`) otherwise.

Description

Call `mxIsInf` to determine whether `value` is equal to infinity or minus infinity. MATLAB software stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable `Inf` is built into the system; you cannot modify it.

Operations that return infinity include:

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

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

Examples

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

- `mxisfinite.c`

See Also

`mxIsFinite`, `mxIsNaN`

Introduced before R2006a

mxIsInt16 (C and Fortran)

Determine whether array represents data as signed 16-bit integers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt16(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

Use `mxIsInt16` to determine whether the specified array represents its real and imaginary data as 16-bit signed integers.

In C, calling `mxIsInt16` is equivalent to calling:

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

In Fortran, calling `mxIsInt16` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt32`, `mxIsInt64`, `mxIsUint8`, `mxIsUint16`, `mxIsUint32`, `mxIsUint64`

Introduced before R2006a

mxIsInt32 (C and Fortran)

Determine whether array represents data as signed 32-bit integers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt32(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

Use `mxIsInt32` to determine whether the specified array represents its real and imaginary data as 32-bit signed integers.

In C, calling `mxIsInt32` is equivalent to calling:

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

In Fortran, calling `mxIsInt32` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt16`, `mxIsInt64`, `mxIsUint8`, `mxIsUint16`, `mxIsUint32`, `mxIsUint64`

Introduced before R2006a

mxIsInt64 (C and Fortran)

Determine whether array represents data as signed 64-bit integers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt64(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

Use `mxIsInt64` to determine whether the specified array represents its real and imaginary data as 64-bit signed integers.

In C, calling `mxIsInt64` is equivalent to calling:

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

In Fortran, calling `mxIsInt64` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt16`, `mxIsInt32`, `mxIsUint8`, `mxIsUint16`, `mxIsUint32`, `mxIsUint64`

Introduced before R2006a

mxIsInt8 (C and Fortran)

Determine whether array represents data as signed 8-bit integers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt8(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

Use `mxIsInt8` to determine whether the specified array represents its real and imaginary data as 8-bit signed integers.

In C, calling `mxIsInt8` is equivalent to calling:

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

In Fortran, calling `mxIsInt8` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt16`, `mxIsInt32`, `mxIsInt64`, `mxIsUint8`, `mxIsUint16`, `mxIsUint32`, `mxIsUint64`

Introduced before R2006a

mxIsLogical (C and Fortran)

Determine whether array is of type mxLogical

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsLogical(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to a logical `mxArray`. Otherwise, it returns logical `0` (`false`).

Description

Use `mxIsLogical` to determine whether MATLAB software treats the data in the `mxArray` as Boolean (logical). If an `mxArray` is logical, MATLAB treats all zeros as meaning `false` and all nonzero values as meaning `true`.

See Also

`mxIsClass`

Topics

"Logical Operations"

Introduced before R2006a

mxIsLogicalScalar (C)

Determine whether scalar array is of type mxLogical

C Syntax

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

Arguments

`array_ptr`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` is of class `mxLogical` and has `1`-by-`1` dimensions. Otherwise, it returns logical `0` (`false`).

Description

Use `mxIsLogicalScalar` to determine whether MATLAB treats the scalar data in the `mxArray` as logical or numerical.

See Also

`mxGetLogicals` | `mxGetScalar` | `mxIsLogical` | `mxIsLogicalScalarTrue`

Topics

"Logical Operations"

Introduced before R2006a

mxIsLogicalScalarTrue (C)

Determine whether scalar array of type mxLogical is true

C Syntax

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

Arguments

`array_ptr`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the value of the `mxArray` logical, scalar element is `true`. Otherwise, it returns logical `0` (`false`).

Description

Use `mxIsLogicalScalarTrue` to determine whether the value of a scalar `mxArray` is true or false.

See Also

`mxGetLogicals` | `mxGetScalar` | `mxIsLogical` | `mxIsLogicalScalar`

Topics

"Logical Operations"

Introduced before R2006a

mxIsNaN (C and Fortran)

Determine whether input is NaN (Not-a-Number)

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsNaN(value)
real*8 value
```

Arguments

`value`

Double-precision, floating-point number to test

Returns

Logical `1` (`true`) if `value` is `NaN` (Not-a-Number), and logical `0` (`false`) otherwise.

Description

Call `mxIsNaN` to determine whether `value` is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. A `NaN` is obtained as a result of mathematically undefined operations such as

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

The system understands a family of bit patterns as representing `NaN`. `NaN` is not a single value; it is a family of numbers that MATLAB software (and other IEEE-compliant applications) uses to represent an error condition or missing data.

Examples

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

- `mxisfinite.c`

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

- `findnz.c`
- `fulltosparse.c`

See Also

`mxIsFinite`, `mxIsInf`

Introduced before R2006a

mxIsNumeric (C and Fortran)

Determine whether array is numeric

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsNumeric(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the array can contain numeric data. The following class IDs represent storage types for arrays that can contain numeric data:

- `mxDOUBLE_CLASS`
- `mxSINGLE_CLASS`
- `mxINT8_CLASS`
- `mxUINT8_CLASS`
- `mxINT16_CLASS`
- `mxUINT16_CLASS`

- mxINT32_CLASS
- mxUINT32_CLASS
- mxINT64_CLASS
- mxUINT64_CLASS

Logical 0 (false) if the array cannot contain numeric data.

Description

Call mxIsNumeric to determine whether the specified array contains numeric data. If the specified array has a storage type that represents numeric data, mxIsNumeric returns logical 1 (true). Otherwise, mxIsNumeric returns logical 0 (false).

Call mxGetClassID to determine the exact storage type.

Examples

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

- phonebook.c

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

- matdemo1.F

See Also

mxGetClassID

Introduced before R2006a

mxIsScalar (C)

Determine whether array is scalar array

C Syntax

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

Arguments

`array_ptr`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` has `1`-by-`1` dimensions. Otherwise, it returns logical `0` (`false`).

Note Only use `mxIsScalar` for `mxArray` classes with IDs documented by `mxClassID`.

Example

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

- `mxisscalar.c`

See Also

`mxClassID` | `mxGetScalar`

Introduced in R2015a

mxIsSingle (C and Fortran)

Determine whether array represents data as single-precision, floating-point numbers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsSingle(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the array stores its data as single-precision, floating-point numbers, and logical `0` (`false`) otherwise.

Description

Use `mxIsSingle` to determine whether the specified array represents its real and imaginary data as single-precision, floating-point numbers.

In C, calling `mxIsSingle` is equivalent to calling:

```
mxGetClassID(pm) == mxSINGLE_CLASS
```

In Fortran, calling `mxIsSingle` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`

Introduced before R2006a

mxIsSparse (C and Fortran)

Determine whether input is sparse array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsSparse(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to a sparse `mxArray`, and logical `0` (`false`) otherwise. A `false` return value means that `pm` points to a full `mxArray` or that `pm` does not point to a valid `mxArray`.

Description

Use `mxIsSparse` to determine whether `pm` points to a sparse `mxArray`. Many routines (for example, `mxGetIr` and `mxGetJc`) require a sparse `mxArray` as input.

Examples

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

- phonebook.c

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

- mxgetnzmax.c
- mxsetdimensions.c
- mxsetdimensionsf.F
- mxsetnzmax.c

See Also

sparse, mxGetIr, mxGetJc, mxCreateSparse

Introduced before R2006a

mxIsStruct (C and Fortran)

Determine whether input is structure array

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsStruct(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to a structure `mxArray`, and logical `0` (`false`) otherwise.

Description

Use `mxIsStruct` to determine whether `pm` points to a structure `mxArray`. Many routines (for example, `mxGetFieldNameByNumber` and `mxSetField`) require a structure `mxArray` as an argument.

Examples

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

- `phonebook.c`

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetFieldNameByNumber`, `mxGetField`, `mxSetField`

Introduced before R2006a

mxIsUint16 (C and Fortran)

Determine whether array represents data as unsigned 16-bit integers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint16(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

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

Description

Use `mxIsUint16` to determine whether the specified `mxArray` represents its real and imaginary data as 16-bit unsigned integers.

In C, calling `mxIsUint16` is equivalent to calling:

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

In Fortran, calling `mxIsUint16` is equivalent to calling:

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

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt16`, `mxIsInt32`, `mxIsInt64`, `mxIsUint8`, `mxIsUint32`, `mxIsUint64`

Introduced before R2006a

mxIsUint32 (C and Fortran)

Determine whether array represents data as unsigned 32-bit integers

C Syntax

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

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint32(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` stores its data as unsigned 32-bit integers, and logical `0` (`false`) otherwise.

Description

Use `mxIsUint32` to determine whether the specified `mxArray` represents its real and imaginary data as 32-bit unsigned integers.

In C, calling `mxIsUint32` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT32_CLASS
```

In Fortran, calling `mxIsUint32` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint32'
```

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt16`, `mxIsInt32`, `mxIsInt64`, `mxIsUint8`, `mxIsUint16`, `mxIsUint64`

Introduced before R2006a

mxIsUint64 (C and Fortran)

Determine whether array represents data as unsigned 64-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsUint64(const mxArray *pm);
```

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint64(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` stores its data as unsigned 64-bit integers, and logical `0` (`false`) otherwise.

Description

Use `mxIsUint64` to determine whether the specified `mxArray` represents its real and imaginary data as 64-bit unsigned integers.

In C, calling `mxIsUint64` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT64_CLASS
```

In Fortran, calling `mxIsUint64` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint64'
```

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt16`, `mxIsInt32`, `mxIsInt64`, `mxIsUint8`, `mxIsUint16`, `mxIsUint32`

Introduced before R2006a

mxIsUint8 (C and Fortran)

Determine whether array represents data as unsigned 8-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsUint8(const mxArray *pm);
```

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint8(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` stores its data as unsigned 8-bit integers, and logical `0` (`false`) otherwise.

Description

Use `mxIsUint8` to determine whether the specified `mxArray` represents its real and imaginary data as 8-bit unsigned integers.

In C, calling `mxIsUint8` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT8_CLASS
```

In Fortran, calling `mxIsUint8` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint8'
```

See Also

`mxIsClass`, `mxGetClassID`, `mxIsInt8`, `mxIsInt16`, `mxIsInt32`, `mxIsInt64`, `mxIsUint16`, `mxIsUint32`, `mxIsUint64`

Introduced before R2006a

mxLogical (C)

Type for logical array

Description

All logical `mxArray`s store their data elements as `mxLogical` rather than as `bool`.

The header file containing this type is:

```
#include "matrix.h"
```

See Also

`mxCreateLogicalArray`

Tips

- For information about data in MATLAB language scripts and functions, see “Data Types”.

Introduced before R2006a

mxMalloc (C and Fortran)

Allocate uninitialized dynamic memory using MATLAB memory manager

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxMalloc(mwSize n);
```

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxMalloc(n)
mwSize n
```

Arguments

`n`

Number of bytes to allocate for `n` greater than `0`

Returns

Pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a MAT or engine standalone application, `mxMalloc` returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and control returns to the MATLAB prompt.

`mxMalloc` is unsuccessful when there is insufficient free heap space.

If you call `mxMalloc` in C with value `n` = `0`, MATLAB returns either `NULL` or a valid pointer.

Description

`mxMalloc` allocates contiguous heap space sufficient to hold `n` bytes. To allocate memory in MATLAB applications, use `mxMalloc` instead of the ANSI C `malloc` function.

In MEX files, but not MAT or engine applications, `mxMalloc` registers the allocated memory with the MATLAB memory manager. When control returns to the MATLAB prompt, the memory manager then automatically frees, or deallocates, this memory.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using the `mxSetPr` function, MATLAB is responsible for freeing the memory.

If you use the data internally, the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX file completes, call `mexMakeMemoryPersistent` after calling this function. If you write a MEX file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX file is cleared.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxmalloc.c`
- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillSetPr.c`

See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxFree`, `mxRealloc`

Introduced before R2006a

mxRealloc (C and Fortran)

Reallocate dynamic memory using MATLAB memory manager

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxRealloc(void *ptr, mwSize size);
```

Fortran Syntax

```
#include "fintrf.h"
mwPointer mxRealloc(ptr, size)
mwPointer ptr
mwSize size
```

Arguments

`ptr`

Pointer to a block of memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

`size`

New size of allocated memory, in bytes.

Returns

Pointer to the start of the reallocated block of memory, if successful. If unsuccessful in a MAT or engine standalone application, `mxRealloc` returns `NULL` in C (`0` in Fortran) and leaves the original memory block unchanged. (Use `mxFree` to free the original memory block). If unsuccessful in a MEX file, the MEX file terminates and control returns to the MATLAB prompt.

`mxRealloc` is unsuccessful when there is insufficient free heap space.

Description

`mxRealloc` changes the size of a memory block that has been allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc`. To allocate memory in MATLAB applications, use `mxRealloc` instead of the ANSI C `realloc` function.

`mxRealloc` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents of the reallocated memory are unchanged up to the smaller of the new and old sizes. The reallocated memory might be in a different location from the original memory, so the returned pointer can be different from `ptr`. If the memory location changes, `mxRealloc` frees the original memory block pointed to by `ptr`.

If `size` is greater than `0` and `ptr` is `NULL` in C (`0` in Fortran), `mxRealloc` behaves like `mxMalloc`. `mxRealloc` allocates a new block of memory of `size` bytes and returns a pointer to the new block.

If `size` is `0` and `ptr` is not `NULL` in C (`0` in Fortran), `mxRealloc` frees the memory pointed to by `ptr` and returns `NULL` in C (`0` in Fortran).

In MEX files, but not MAT or engine applications, `mxRealloc` registers the allocated memory with the MATLAB memory manager. When control returns to the MATLAB prompt, the memory manager then automatically frees, or deallocates, this memory.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using the `mxSetPr` function, MATLAB is responsible for freeing the memory.

If you use the data internally, the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX file completes, call `mexMakeMemoryPersistent` after calling this function. If you write a MEX file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX file is cleared.

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetnzmax.c

See Also

mexAtExit, mexMakeArrayPersistent, mexMakeMemoryPersistent, mxCalloc, mxDestroyArray, mxFree, mxMalloc

Introduced before R2006a

mxRemoveField (C and Fortran)

Remove field from structure array

C Syntax

```
#include "matrix.h"
void mxRemoveField(mxArray *pm, int fieldnumber);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxRemoveField(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

Number of the field you want to remove. In C, to remove the first field, set `fieldnumber` to `0`; to remove the second field, set `fieldnumber` to `1`; and so on. In Fortran, to remove the first field, set `fieldnumber` to `1`; to remove the second field, set `fieldnumber` to `2`; and so on.

Description

Call `mxRemoveField` to remove a field from a structure array. If the field does not exist, nothing happens. This function does not destroy the field values. To destroy the actual field values, call `mxRemoveField` and then call `mxDestroyArray`.

Consider a MATLAB structure initialized to:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field number `0` represents the field `name`; field number `1` represents field `billing`; field number `2` represents field `test`. In Fortran, the field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`.

See Also

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

Introduced before R2006a

mxSetCell (C and Fortran)

Set contents of cell array

C Syntax

```
#include "matrix.h"
void mxSetCell(mxArray *pm, mwIndex index, mxArray *value);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetCell(pm, index, value)
mwPointer pm, value
mwIndex index
```

Arguments

`pm`

Pointer to a cell `mxArray`

`index`

Index from the beginning of the `mxArray`. Specify the number of elements between the first cell of the `mxArray` and the cell you want to set. The easiest way to calculate `index` in a multidimensional cell array is to call `mxCalcSingleSubscript`.

`value`

Pointer to new value for the cell. You can put an `mxArray` of any type into a cell. You can even put another cell `mxArray` into a cell.

Description

Call `mxSetCell` to put the designated `value` into a particular cell of a cell `mxArray`.

Note Inputs to a MEX-file are constant read-only `mxArray`s. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxDestroyArray` on the pointer returned by `mxGetCell` before you call `mxSetCell`.

Examples

See the following examples in *matlabroot*`/extern/examples/refbook`.

- `phonebook.c`

See the following examples in *matlabroot*`/extern/examples/mx`.

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`

See Also

`mxCreateCellArray`, `mxCreateCellMatrix`, `mxGetCell`, `mxIsCell`, `mxDestroyArray`

Introduced before R2006a

mxSetClassName (C)

Structure array to MATLAB object array

Note Use `mxSetClassName` for classes defined without a `classdef` statement.

C Syntax

```
#include "matrix.h"
int mxSetClassName(mxArray *array_ptr, const char *classname);
```

Arguments

`array_ptr`

Pointer to an `mxArray` of class `mxSTRUCT_CLASS`

`classname`

Object class to which to convert `array_ptr`

Returns

`0` if successful, and nonzero otherwise. One cause of failure is that `array_ptr` is not a structure `mxArray`. Call `mxIsStruct` to determine whether `array_ptr` is a structure.

Description

`mxSetClassName` converts a structure array to an object array, to be saved later to a MAT-file. MATLAB does not register or validate the object until it is loaded by 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`, `mxIsStruct`

Introduced before R2006a

mxSetData (C and Fortran)

Set pointer to real numeric data elements in array

C Syntax

```
#include "matrix.h"
void mxSetData(mxArray *pm, void *pr);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetData(pm, pr)
mwPointer pm, pr
```

Arguments

`pm`

Pointer to an `mxArray`

`pr`

Pointer to an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this memory. Do not use the ANSI C `calloc` function, which can cause memory alignment issues leading to program termination.

Description

`mxSetData` is like `mxSetPr`, except that in C, its second argument is a `void *`. Use this function on numeric arrays with contents other than `double`.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetData` before you call `mxSetData`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillSetData.c`

See Also

`mxCalloc`, `mxFree`, `mxGetData`, `mxSetPr`

Introduced before R2006a

mxSetDimensions (C and Fortran)

Modify number of dimensions and size of each dimension

C Syntax

```
#include "matrix.h"
int mxSetDimensions(mxArray *pm, const mwSize *dims, mwSize ndim);
```

Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetDimensions(pm, dims, ndim)
mwPointer pm
mwSize ndim
mwSize dims(ndim)
```

Arguments

`pm`

Pointer to an `mxArray`

`dims`

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to `5` and `dims[1]` to `7` establishes a `5`-by-`7` `mxArray`. In Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a `5`-by-`7` `mxArray`. In most cases, there are `ndim` elements in the `dims` array.

`ndim`

Number of dimensions

Returns

`0` on success, and `1` on failure. `mxSetDimensions` allocates heap space to hold the input size array. So it is possible (though unlikely) that increasing the number of dimensions can cause the system to run out of heap space.

Description

Call `mxSetDimensions` to reshape an existing `mxArray`. `mxSetDimensions` is like `mxSetM` and `mxSetN`; however, `mxSetDimensions` provides greater control for reshaping an `mxArray` that has more than two dimensions.

`mxSetDimensions` does not allocate or deallocate any space for the `pr` or `pi` arrays. So, if your call to `mxSetDimensions` increases the number of elements in the `mxArray`, enlarge the `pr` (and `pi`, if it exists) arrays accordingly.

If your call to `mxSetDimensions` reduces the number of elements in the `mxArray`, you can optionally reduce the size of the `pr` and `pi` arrays using `mxRealloc`.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array has the dimensions `4`-by-`1`-by-`7`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetdimensions.c`
- `mxsetdimensionsf.F`

See Also

`mxGetNumberOfDimensions`, `mxSetM`, `mxSetN`, `mxRealloc`

Introduced before R2006a

mxSetField (C and Fortran)

Set field value in structure array, given index and field name

C Syntax

```
#include "matrix.h"
void mxSetField(mxArray *pm, mwIndex index,
  const char *fieldname, mxArray *pvalue);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetField(pm, index, fieldname, pvalue)
mwPointer pm, pvalue
mwIndex index
character*(*) fieldname
```

Arguments

`pm`

Pointer to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

`index`

Index of an element in the array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for details on calculating an index.

`fieldname`

Name of a field in the structure. The field must exist in the structure. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to determine existing field names.

`pvalue`

Pointer to an `mxArray` containing the data you want to assign to `fieldname`.

Description

Use `mxSetField` to assign the contents of `pvalue` to the field `fieldname` of element `index`.

If you want to replace the contents of `fieldname`, first free the memory of the existing data. Use the `mxGetField` function to get a pointer to the field, call `mxDestroyArray` on the pointer, then call `mxSetField` to assign the new value.

You cannot assign `pvalue` to more than one field in a structure or to more than one element in the `mxArray`. If you want to assign the contents of `pvalue` to multiple fields, use the `mxDuplicateArray` function to make copies of the data then call `mxSetField` on each copy.

To free memory for structures created using this function, call `mxDestroyArray` only on the structure array. Do not call `mxDestroyArray` on the array `pvalue` points to. If you do, MATLAB attempts to free the same memory twice, which can corrupt memory.

Note Inputs to a MEX-file are constant read-only `mxArray`s. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxcreatestructarray.c`

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetFieldByNumber`, `mxDestroyArray`, `mxCalcSingleSubscript`

Alternatives

C Language

In C, you can replace the statements:

```
field_num = mxGetFieldNumber(pa, "fieldname");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

with a call to `mxSetField`:

```
mxSetField(pa, index, "fieldname", new_value_pa);
```

Fortran Language

In Fortran, you can replace the statements:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

with a call to `mxSetField`:

```
mxSetField(pm, index, 'fieldname', newvalue)
```

Introduced before R2006a

mxSetFieldByNumber (C and Fortran)

Set field value in structure array, given index and field number

C Syntax

```
#include "matrix.h"
void mxSetFieldByNumber(mxArray *pm, mwIndex index,
  int fieldnumber, mxArray *pvalue);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetFieldByNumber(pm, index, fieldnumber, pvalue)
mwPointer pm, pvalue
mwIndex index
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for details on calculating an index.

`fieldnumber`

Position of the field in the structure. The field must exist in the structure.

In C, the first field within each element has a `fieldnumber` of `0`. The `fieldnumber` of the last is `N-1`, where `N` is the number of fields.

In Fortran, the first field within each element has a `fieldnumber` of `1`. The `fieldnumber` of the last is `N`, where `N` is the number of fields.

`pvalue`

Pointer to the `mxArray` containing the data you want to assign.

Description

Use `mxSetFieldByNumber` to assign the contents of `pvalue` to the field specified by `fieldnumber` of element `index`. `mxSetFieldByNumber` is like `mxSetField`; however, the function identifies the field by position number, not by name.

If you want to replace the contents at `fieldnumber`, first free the memory of the existing data. Use the `mxGetFieldByNumber` function to get a pointer to the field, call `mxDestroyArray` on the pointer, then call `mxSetFieldByNumber` to assign the new value.

You cannot assign `pvalue` to more than one field in a structure or to more than one element in the `mxArray`. If you want to assign the contents of `pvalue` to multiple fields, use the `mxDuplicateArray` function to make copies of the data then call `mxSetFieldByNumber` on each copy.

To free memory for structures created using this function, call `mxDestroyArray` only on the structure array. Do not call `mxDestroyArray` on the array `pvalue` points to. If you do, MATLAB attempts to free the same memory twice, which can corrupt memory.

Note Inputs to a MEX-file are constant read-only `mxArray`s. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

Alternatives

C Language

In C, calling:

```
mxSetField(pa, index, "field_name", new_value_pa);
```

is equivalent to calling:

```
field_num = mxGetFieldNumber(pa, "field_name");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

Fortran Language

In Fortran, calling:

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

Examples

See the following examples in `matlabroot/extern/examples/mx`.

- `mxcreatestructarray.c`

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxDestroyArray`, `mxCalcSingleSubscript`

Introduced before R2006a

mxSetImagData (C and Fortran)

Set pointer to imaginary data elements in array

C Syntax

```
#include "matrix.h"
void mxSetImagData(mxArray *pm, void *pi);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetImagData(pm, pi)
mwPointer pm, pi
```

Arguments

`pm`

Pointer to an `mxArray`

`pi`

Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this memory. Do not use the ANSI C `calloc` function, which can cause memory alignment issues leading to program termination. If `pi` points to static memory, memory errors will result when the array is destroyed.

Description

`mxSetImagData` is like `mxSetPi`, except that in C, its `pi` argument is a `void *`. Use this function on numeric arrays with contents other than `double`.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetImagData` before you call `mxSetImagData`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`

See Also

`mxCalloc`, `mxFree`, `mxGetImagData`, `mxSetPi`

Introduced before R2006a

mxSetIr (C and Fortran)

IR array of sparse array

C Syntax

```
#include "matrix.h"
void mxSetIr(mxArray *pm, mwIndex *ir);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetIr(pm, ir)
mwPointer pm, ir
```

Arguments

`pm`

Pointer to a sparse `mxArray`

`ir`

Pointer to the `ir` array. The `ir` array must be sorted in column-major order.

Description

Use `mxSetIr` to specify the `ir` array of a sparse `mxArray`. The `ir` array is an array of integers; the length of the `ir` array equals the value of `nzmax`, the storage allocated for the sparse array, or `nnz`, the number of nonzero matrix elements.

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 that you create a `7`-by-`3` sparse `mxArray` named `Sparrow` containing six nonzero elements by typing:

```
Sparrow = zeros(7,3);
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The `pr` array holds the real data for the sparse matrix, which in `Sparrow` is the five 1s and the one 2. If there is any nonzero imaginary data, it is in a `pi` array.

Subscript	ir	pr	jc	Comments
`(2,1)`	`1`	`1`	`0`	Column 1; `ir` is 1 because row is 2.
`(5,1)`	`4`	`1`	`2`	Column 1; `ir` is 4 because row is 5.
`(3,2)`	`2`	`1`	`3`	Column 2; `ir` is 2 because row is 3.
`(2,3)`	`1`	`2`	`6`	Column 3; `ir` is 1 because row is 2.
`(5,3)`	`4`	`1`		Column 3; `ir` is 4 because row is 5.
`(6,3)`	`5`	`1`		Column 3; `ir` is 5 because row is 6.

Notice how each element of the `ir` array is always 1 less than the row of the corresponding nonzero element. For instance, the first nonzero element is in row 2; therefore, the first element in `ir` is 1 (that is, 2 – 1). The second nonzero element is in row 5; therefore, the second element in `ir` is 4 (5 – 1).

The `ir` array must be in column-major order. The `ir` array must define the row positions in column 1 (if any) first, then the row positions in column 2 (if any) second, and so on, through column N. Within each column, row position 1 must appear before row position 2, and so on.

`mxSetIr` does not sort the `ir` array for you; you must specify an `ir` array that is already sorted.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetIr` before you call `mxSetIr`.

Examples

See the following examples in *matlabroot*/extern/examples/mx.

- mxsetnzmax.c

See the following examples in *matlabroot*/extern/examples/mex.

- explore.c

See Also

mxCreateSparse, mxGetIr, mxGetJc, mxSetJc, mxFree, nzmax, nnz

Introduced before R2006a

mxSetJc (C and Fortran)

JC array of sparse array

C Syntax

```
#include "matrix.h"
void mxSetJc(mxArray *pm, mwIndex *jc);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetJc(pm, jc)
mwPointer pm, jc
```

Arguments

`pm`

Pointer to a sparse `mxArray`

`jc`

Pointer to the `jc` array

Description

Use `mxSetJc` to specify a new `jc` array for a sparse `mxArray`. The `jc` array is an integer array having `n+1` elements, where `n` is the number of columns in the sparse `mxArray`.

If the `j`th column of the sparse `mxArray` has any nonzero elements:

- `jc[j]` is the index in `ir`, `pr`, and `pi` (if it exists) of the first nonzero element in the `j`th column.
- `jc[j+1]-1` is the index of the last nonzero element in the `j`th column.

- For the `j`th column of the sparse matrix, `jc[j]` is the total number of nonzero elements in all preceding columns.

The number of nonzero elements in the `j`th column of the sparse `mxArray` is:

```
jc[j+1] - jc[j];
```

For the `j`th column of the sparse `mxArray`, `jc[j]` is the total number of nonzero elements in all preceding columns. The last element of the `jc` array, `jc[`number of columns`]`, is equal to `nnz`, which is the number of nonzero elements in the entire sparse `mxArray`.

For example, consider a `7`-by-`3` sparse `mxArray` named `Sparrow` containing six nonzero elements, created by typing:

```
Sparrow = zeros(7,3);
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The following table lists the contents of the `ir`, `jc`, and `pr` arrays.

Subscript	ir	pr	jc	Comment
`(2,1)`	1	1	0	Column 1 contains two nonzero elements, with rows designated by `ir[0]` and `ir[1]`
`(5,1)`	4	1	2	Column 2 contains one nonzero element, with row designated by `ir[2]`
`(3,2)`	2	1	3	Column 3 contains three nonzero elements, with rows designated by `ir[3]`,`ir[4]`, and `ir[5]`
`(2,3)`	1	2	6	There are six nonzero elements in all.
`(5,3)`	4	1		
`(6,3)`	5	1		

As an example of a much sparser `mxArray`, consider a `1000`-by-`8` sparse `mxArray` named `Spacious` containing only three nonzero elements. The `ir`, `pr`, and `jc` arrays contain the values listed in this table.

Subscript	ir	pr	jc	Comment
`(73,2)`	`72`	`1`	`0`	Column 1 contains no nonzero elements.
`(50,3)`	`49`	`1`	`0`	Column 2 contains one nonzero element, with row designated by `ir[0]`.
`(64,5)`	`63`	`1`	`1`	Column 3 contains one nonzero element, with row designated by `ir[1]`.
			`2`	Column 4 contains no nonzero elements.
			`2`	Column 5 contains one nonzero element, with row designated by `ir[2]`.
			`3`	Column 6 contains no nonzero elements.
			`3`	Column 7 contains no nonzero elements.
			`3`	Column 8 contains no nonzero elements.
			`3`	There are three nonzero elements in all.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetJc` before you call `mxSetJc`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/mex`.

- `explore.c`

See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetIr`, `mxFree`

Introduced before R2006a

mxSetM (C and Fortran)

Set number of rows in array

C Syntax

```
#include "matrix.h"
void mxSetM(mxArray *pm, mwSize m);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetM(pm, m)
mwPointer pm
mwSize m
```

Arguments

`pm`

Pointer to an `mxArray`

`m`

Number of rows

Description

Call `mxSetM` to set the number of rows in the specified `mxArray`. The term *rows* means the first dimension of an `mxArray`, regardless of the number of dimensions. Call `mxSetN` to set the number of columns.

You typically use `mxSetM` to change the shape of an existing `mxArray`. The `mxSetM` function does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. So, if your calls to `mxSetM` and `mxSetN` increase the number of elements in the `mxArray`, enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays. Call `mxRealloc` to enlarge them.

If calling `mxSetM` and `mxSetN` reduces the number of elements in the `mxArray`, you might want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays to use heap space more efficiently. However, reducing the size is not mandatory.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `sincall.c`
- `sincall.F`

See Also

`mxGetM`, `mxGetN`, `mxSetN`, `mxRealloc`

Introduced before R2006a

mxSetN (C and Fortran)

Set number of columns in array

C Syntax

```
#include "matrix.h"
void mxSetN(mxArray *pm, mwSize n);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetN(pm, n)
mwPointer pm
mwSize n
```

Arguments

`pm`

Pointer to an `mxArray`

`n`

Number of columns

Description

Call `mxSetN` to set the number of columns in the specified `mxArray`. The term *columns* always means the second dimension of a matrix. Calling `mxSetN` forces an `mxArray` to have two dimensions. For example, if `pm` points to an `mxArray` having three dimensions, calling `mxSetN` reduces the `mxArray` to two dimensions.

You typically use `mxSetN` to change the shape of an existing `mxArray`. The `mxSetN` function does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. So, if your calls to `mxSetN` and `mxSetM` increase the number of elements in the `mxArray`, enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays.

If calling `mxSetM` and `mxSetN` reduces the number of elements in the `mxArray`, you might want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays to use heap space more efficiently. However, reducing the size is not mandatory.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetdimensions.c`

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `sincall.c`
- `sincall.F`

See Also

`mxGetM`, `mxGetN`, `mxSetM`

Introduced before R2006a

mxSetNzmax (C and Fortran)

Set storage space for nonzero elements

C Syntax

```
#include "matrix.h"
void mxSetNzmax(mxArray *pm, mwSize nzmax);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetNzmax(pm, nzmax)
mwPointer pm
mwSize nzmax
```

Arguments

`pm`

Pointer to a sparse `mxArray`.

`nzmax`

Number of elements `mxCreateSparse` should allocate to hold the arrays pointed to by `ir`, `pr`, and `pi` (if it exists). Set `nzmax` greater than or equal to the number of nonzero elements in the `mxArray`, but set it to be less than or equal to the number of rows times the number of columns. If you specify an `nzmax` value of `0`, `mxSetNzmax` sets the value of `nzmax` to `1`.

Description

Use `mxSetNzmax` to assign a new value to the `nzmax` field of the specified sparse `mxArray`. The `nzmax` field holds the maximum number of nonzero elements in the sparse `mxArray`.

The number of elements in the `ir`, `pr`, and `pi` (if it exists) arrays must be equal to `nzmax`. Therefore, after calling `mxSetNzmax`, you must change the size of the `ir`, `pr`, and `pi` arrays. To change the size of one of these arrays:

1. Call `mxRealloc` with a pointer to the array, setting the size to the new value of `nzmax`.
2. Call the appropriate `mxSet` routine (`mxSetIr`, `mxSetPr`, or `mxSetPi`) to establish the new memory area as the current one.

Ways to determine how large to make `nzmax` are:

- Set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space.
- Make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetnzmax.c`

See Also

`mxGetNzmax`, `mxRealloc`

Introduced before R2006a

mxSetPi (C and Fortran)

Set new imaginary data elements in array of type DOUBLE

C Syntax

```
#include "matrix.h"
void mxSetPi(mxArray *pm, double *pi);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetPi(pm, pi)
mwPointer pm, pi
```

Arguments

`pm`

Pointer to a full (nonsparse) `mxArray`

`pi`

Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this memory. Do not use the ANSI C `calloc` function, which can cause memory alignment issues leading to program termination. If `pi` points to static memory, memory leaks and other memory errors might result.

Description

Use `mxSetPi` to set the imaginary data of the specified `mxArray`.

Most `mxCreate*` functions optionally allocate heap space to hold imaginary data. If you tell an `mxCreate*` function to allocate heap space—for example, by setting the `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran) or by setting `pi` to a non-`NULL` value in

C (a nonzero value in Fortran)—you do not ordinarily use `mxSetPi` to initialize the created `mxArray`'s imaginary elements. Rather, you call `mxSetPi` to replace the initial imaginary values with new ones.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetPi` before you call `mxSetPi`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxisfinite.c`
- `mxsetnzmax.c`

See Also

`mxGetPi`, `mxGetPr`, `mxSetImagData`, `mxSetPr`, `mxFree`

Introduced before R2006a

mxSetPr (C and Fortran)

Set new real data elements in array of type DOUBLE

C Syntax

```
#include "matrix.h"
void mxSetPr(mxArray *pm, double *pr);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetPr(pm, pr)
mwPointer pm, pr
```

Arguments

`pm`

Pointer to a full (nonsparse) `mxArray`

`pr`

Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this memory. Do not use the ANSI C `calloc` function, which can cause memory alignment issues leading to program termination. If `pr` points to static memory, memory leaks and other memory errors can result.

Description

Use `mxSetPr` to set the real data of the specified `mxArray`.

All `mxCreate*` calls allocate heap space to hold real data. Therefore, you do not ordinarily use `mxSetPr` to initialize the real elements of a freshly created `mxArray`. Rather, you call `mxSetPr` to replace the initial real values with new ones.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetPr` before you call `mxSetPr`.

Examples

See the following examples in *`matlabroot`*`/extern/examples/refbook`.

- `arrayFillSetPr.c`

See the following examples in *`matlabroot`*`/extern/examples/mx`.

- `mxsetnzmax.c`

See Also

`mxGetPi`, `mxGetPr`, `mxSetData`, `mxSetPi`, `mxFree`

Introduced before R2006a

mxSetProperty (C and Fortran)

Set value of public property of MATLAB object

C Syntax

```
#include "matrix.h"
void mxSetProperty(mxArray *pa, mwIndex index,
  const char *propname, const mxArray *value);
```

Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetProperty(pa, index, propname, value)
mwPointer pa, value
mwIndex index
character*(*) propname
```

Arguments

`pa`

Pointer to an `mxArray` which is an object.

`index`

Index of the desired element of the object array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`propname`

Name of the property whose value you are assigning.

value

Pointer to the `mxArray` you are assigning.

Description

Use `mxSetProperty` to assign a `value` to the specified property. In pseudo-C terminology, `mxSetProperty` performs the assignment:

```
pa[index].propname = value;
```

Property `propname` must be an existing, public property and `index` must be within the bounds of the `mxArray`. To test the index value, use `mxGetNumberOfElements` or `mxGetM` and `mxGetN` functions.

`mxSetProperty` makes a copy of the value before assigning it as the new property value. Making a copy might be a concern if the property uses a large amount of memory. There must be sufficient memory (in the heap) to hold the copy of the value.

Limitations

- `mxSetProperty` is not supported for standalone applications, such as applications built with the MATLAB engine API.

See Also

`mxGetProperty`

Introduced in R2008a