

MATLAB®

External Interfaces



MATLAB®

R2019b

 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.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® External Interfaces

© COPYRIGHT 1984-2019 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

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

Revision History

December 1996	First printing	New for MATLAB 5 (release 8)
July 1997	Online only	Revised for MATLAB 5.1 (Release 9)
January 1998	Second printing	Revised for MATLAB 5.2 (Release 10)
October 1998	Third printing	Revised for MATLAB 5.3 (Release 11)
November 2000	Fourth printing	Revised and renamed 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)
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 for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised 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 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)
March 2018	Online only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online only	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)

1 External Programming Languages and Systems

Integrate MATLAB with External Programming Languages and Systems	1-2
Call C/C++ Code from MATLAB	1-2
Use Objects from Other Programming Languages in MATLAB	1-3
Call MATLAB from Another Programming Language	1-3
Call Your Functions as MATLAB Functions	1-4
Communicate with Web Services	1-4
Custom Linking to Required API Libraries	1-6
C++ MEX Functions	1-6
C++ Engine Applications	1-7
C MEX Functions	1-7
C Engine Applications	1-8
C MAT-File Applications	1-9

2 External Data Interface (EDI)

Create Arrays with C++ MATLAB Data API	2-2
Create Arrays	2-2
Operate on Each Element in an Array	2-3
Copy C++ MATLAB Data Arrays	2-5
Avoid Unnecessary Data Copying	2-5
C++ Cell Arrays	2-7

Access C++ Data Array Container Elements	2-9
Modify By Reference	2-9
Copy Data from Container	2-9
Operate on C++ Arrays Using Visitor Pattern	2-11
Dispatch on Array or Array Reference	2-11
Overloading operator()	2-11
Visitor Class to Display Contents of Cell Array	2-12
Visitor Class to Modify Contents of Cell Array	2-14
MATLAB Data API Exceptions	2-17
matlab::data::CanOnlyUseOneStringIndexException	2-17
matlab::data::CantAssignArrayToThisArrayException	2-18
matlab::data::CantIndexIntoEmptyArrayException	2-18
matlab::data::DuplicateFieldNameInStructArrayException ..	2-18
matlab::data::FailedToLoadLibMatlabDataArrayException ..	2-18
matlab::data::FailedToResolveSymbolException	2-18
matlab::data::InvalidArrayIndexException	2-18
matlab::data::InvalidDimensionsInSparseArrayException ...	2-18
matlab::data::InvalidFieldNameException	2-18
matlab::data::MustSpecifyClassNameException	2-19
matlab::data::NonAsciiCharInRequestedAsciiOutputException	
.....	2-19
matlab::data::NonAsciiCharInInputDataException	2-19
matlab::data::InvalidArrayTypeException	2-19
matlab::data::NotEnoughIndicesProvidedException	2-19
matlab::data::StringIndexMustBeLastException	2-19
matlab::data::StringIndexNotValidException	2-19
matlab::data::SystemErrorException	2-20
matlab::data::TooManyIndicesProvidedException	2-20
matlab::data::TypeMismatchException	2-20
matlab::data::WrongNumberOfEnumsSuppliedException ...	2-20
matlab::data::InvalidMemoryLayout	2-20
matlab::data::InvalidDimensionsInRowMajorArrayException	
.....	2-20
matlab::data::NumberOfElementsExceedsMaximumException	
.....	2-20
MATLAB Data API Types	2-21
matlab::data::ArrayDimensions	2-21
matlab::data::Enumeration	2-21
matlab::data::MATLABString	2-21
matlab::data::ObjectArray	2-21
matlab::data::String	2-22

matlab::data::Struct	2-22
buffer_ptr_t and buffer_deleter_t	2-22
iterator	2-22
const_iterator	2-22
reference	2-22
const_reference	2-22
Reference Types	2-22

Using Java Libraries from MATLAB

3

Getting Started with Java Libraries	3-2
Platform Support for JVM Software	3-2
Learn More About Java Programming Language	3-2
Call Java Method	3-4
Choose Class Constructor to Create Java Object	3-4
Shorten Class Name	3-4
Create Array List	3-4
Pass MATLAB Data to add Method	3-5
Add Elements to ArrayList	3-5
Java Objects Are References in MATLAB	3-5
Use ArrayList Object in MATLAB	3-6
Simplify Java Class Names Using import Function	3-8
Java Class Path	3-10
Static Path	3-12
Create javaclasspath.txt File	3-12
Add Individual (Unpackaged) Classes	3-13
Add Packages	3-13
Add JAR File Classes	3-13
Load Java Class Definitions	3-15
Locate Native Method Libraries	3-16
Load Class Using Java Class.forName Method	3-17

Dynamic Path	3-18
Call Method in Your Own Java Class	3-19
How MATLAB Represents Java Arrays	3-21
Array Indexing	3-21
Shape of Java Arrays	3-21
Interpret Size of Java Arrays	3-22
Interpret Number of Dimensions of Java Arrays	3-23
Display Java Vector	3-23
Create Array of Java Objects	3-24
Create Array of Primitive Java Types	3-24
Access Elements of Java Array	3-26
MATLAB Array Indexing	3-26
Single Subscript Indexing	3-26
Colon Operator Indexing	3-27
Using END in a Subscript	3-29
Converting Object Array Elements to MATLAB Types	3-29
Assign Values to Java Array	3-31
Single Subscript Indexing Assignment	3-31
Linear Array Assignment	3-32
Empty Matrix Assignment	3-33
Subscripted Deletion	3-33
Concatenate Java Arrays	3-35
Two-Dimensional Horizontal Concatenation	3-35
Vector Concatenation	3-36
Create Java Array References	3-39
Create Copy of Java Array	3-41
Construct and Concatenate Java Objects	3-43
Create Java Object	3-43
Concatenate Objects of Same Class	3-43
Concatenate Objects of Unlike Classes	3-43
Save and Load Java Objects to MAT-Files	3-45

Data Fields of Java Objects	3-46
Access Public and Private Data	3-46
Display Public Data Fields of Java Object	3-47
Access Static Field Data	3-47
Determine Class of Java Object	3-49
Method Information	3-50
Display Method Names	3-50
Display Method Signatures	3-50
Display Syntax in Figure Window	3-51
Determine What Classes Define a Method	3-52
Java Methods That Affect MATLAB Commands	3-54
Changing the Effect of disp and display	3-54
Changing the Effect of isequal	3-54
Changing the Effect of double, string, and char	3-55
How MATLAB Handles Undefined Methods	3-56
Avoid Calling Java main Methods in MATLAB	3-57
Pass Data to Java Methods	3-58
MATLAB Type to Java Type Mapping	3-58
How Array Dimensions Affect Conversion	3-59
Convert Numbers to Integer Arguments	3-60
Pass String Arguments	3-60
Pass Java Objects	3-61
Pass Empty Matrices, Nulls, and Missing Values	3-62
Overloaded Methods	3-62
Handle Data Returned from Java Methods	3-64
Primitive Return Types	3-64
java.lang.Object Return Types	3-65
Functions to Convert Java Objects to MATLAB Types	3-66
Java Heap Memory Preferences	3-70

Read and Write MATLAB MAT-Files in C/C++ and Fortran

4

Choosing Applications to Read and Write MATLAB MAT-Files	4-2
Custom Applications to Access MAT-Files	4-4
Why Write Custom Applications?	4-4
MAT-File Interface Library	4-5
Exchanging Data Files Between Platforms	4-6
MAT-File API Library and Include Files	4-7
MAT-File API Include Files	4-7
MAT-File API Libraries	4-7
Example Files	4-8
What You Need to Build Custom Applications	4-9
Copy External Data into MAT-File Format with Standalone Programs	4-10
Overview of matimport.c Example	4-10
Declare Variables for External Data	4-11
Create mxArray Variables	4-11
Create MATLAB Variable Names	4-12
Read External Data into mxArray Data	4-12
Create and Open MAT-File	4-13
Write mxArray Data to File	4-13
Clean Up	4-13
Build the Application	4-13
Create the MAT-File	4-13
Import Data into MATLAB	4-14
Create MAT-File in C or C++	4-15
Create MAT-File in C	4-15
Create MAT-File in C++	4-15
Read MAT-File in C/C++	4-17
Create MAT-File in Fortran	4-19
Read MAT-File in Fortran	4-20

Work with mxArray	4-21
Read Structures from a MAT-File	4-21
Read Cell Arrays from a MAT-File	4-22
Table of MAT-File Source Code Files	4-24
Build on macOS and Linux Operating Systems	4-26
Set Run-Time Library Path	4-26
Build Application	4-27
Build on Windows Operating Systems	4-29
Share MAT-File Applications	4-30

Calling Functions in C++ Shared Libraries from MATLAB

5

System Requirements	5-2
Compiler Dependencies	5-2
Set Run-Time Library Path for C++ Interface	5-3
Temporarily Set Run-Time Library Path	5-3
Steps to Publish a MATLAB C++ Library Interface	5-5
One-Step Build	5-6
Publish Interface to Shared C++ Library on Windows	5-7
Generate Definition File	5-7
Define Missing Constructs	5-8
Build Library Interface	5-9
Test the Interface	5-9
Publish Interface to Shared C++ Library on Linux	5-11
Call Functions in Windows Interface to C++ Shared Library	5-13
Set Paths	5-13
View Help	5-13
Call Library Functions	5-14

Call Functions in Linux Interface to C++ Shared Library . . .	5-16
Set Paths	5-16
View Help	5-16
Call Library Functions	5-17
Publish Interface to Header-Only C++ Library	5-19
Verify Selected C++ Compiler	5-19
Copy Example Header File	5-19
Generate Definition File	5-19
Define Missing Information for C++ Function getName	5-19
Build Interface and Add to MATLAB Path	5-20
View Contents of Library	5-20
Test Functions	5-21
Modify Help Text	5-22
Distribute Interface	5-22
Display Help for MATLAB Interface to C++ Library	5-23
Publish Modified Help Text	5-25
Define MATLAB Interface to C++ Library	5-27
Code Suggestions for Parameters	5-27
DIRECTION Parameter	5-27
SHAPE Parameter	5-28
MLTYPE Parameter	5-36
Build Library and Review Contents	5-37
Sample C++ Library Definition File	5-38
Call Functions in C++ Shared Library	5-43
Set Path	5-43
Call Function	5-43
Limitations to C/C++ Support	5-44
Data Types and Language Features Dropped From Interface	5-44
Language Features Not Supported	5-45
Unsupported Usage	5-45
C++ Names That Are Invalid in MATLAB	5-47
Troubleshooting C++ Interface Issues	5-48
Why Is a Function Missing from the Definition File?	5-48

MATLAB Did Not Create .mlx Definition File	5-48
Cannot Open .mlx Definition File	5-48
MATLAB to C++ Data Type Mapping	5-49
Non-Fixed-Width Integer Types	5-49
String and Character Types	5-50
User-Defined Types	5-51
nullptr Return Types	5-51
Unsupported Data Types	5-51
Handling Exceptions	5-53
Errors Parsing Header Files on macOS	5-54
Build Error Due to Compile-Time Checks	5-56

Calling Functions in C Shared Libraries from MATLAB

6

Call C Functions in Shared Libraries	6-2
Invoke Library Functions	6-4
View Library Functions	6-5
Load and Unload Library	6-6
Limitations to Shared Library Support	6-7
MATLAB Supports C Library Routines	6-7
Loading C++ Libraries	6-7
Limitations Using printf Function	6-7
Bit Fields	6-8
Enum Declarations	6-8
Unions Not Supported	6-9
Compiler Dependencies	6-9
Limitations Using Pointers	6-9
Functions with Variable Number of Input Arguments Not Supported	6-10

Limitations Using Structures	6-11
MATLAB Returns Pointers to Structures	6-11
Structure Cannot Contain Pointers to Other Structures	6-11
Requirements for MATLAB Structure Arguments	6-11
Requirements for C struct Field Names	6-11
Loading Library Errors	6-13
No Matching Signature Error	6-14
MATLAB Terminates Unexpectedly When Calling Function in Shared Library	6-15
Pass Arguments to Shared C Library Functions	6-16
C and MATLAB Equivalent Types	6-16
How MATLAB Displays Function Signatures	6-19
NULL Pointer	6-19
Manually Convert Data Passed to Functions	6-20
Shared Library shrllibsample	6-22
Pass String Arguments Examples	6-23
stringToUpper Function	6-23
Convert MATLAB Character Array to Uppercase	6-23
Pass Structures Examples	6-25
addStructFields and addStructByRef Functions	6-25
Add Values of Fields in Structure	6-26
Preconvert MATLAB Structure Before Adding Values	6-27
Autoconvert Structure Arguments	6-28
Pass Pointer to Structure	6-28
Pass Enumerated Types Examples	6-31
readEnum Function	6-31
Display Enumeration Values	6-32
Pass Pointers Examples	6-33
multDoubleRef Function	6-33
Pass Pointer of Type double	6-33
Create Pointer Offset from Existing lib.pointer Object	6-34
Multilevel Pointers	6-35
allocateStruct and deallocateStruct Functions	6-35
Pass Multilevel Pointer	6-36

Return Array of Strings	6-36
Pass Arrays Examples	6-38
print2darray Function	6-38
Convert MATLAB Array to C-Style Dimensions	6-38
multDoubleArray Function	6-39
Preserve 3-D MATLAB Array	6-40
Iterate Through an Array	6-42
Create Cell Array from lib.pointer Object	6-42
Perform Pointer Arithmetic on Structure Array	6-43
Pointer Arguments	6-45
Pointer Arguments in C Functions	6-45
Put String into Void Pointer	6-45
Memory Allocation for External Library	6-46
Structure Arguments	6-48
Structure Argument Requirements	6-48
Find Structure Field Names	6-48
Strategies for Passing Structures	6-48
Explore libstruct Objects	6-50
MATLAB Prototype Files	6-51
When to Use Prototype Files	6-51
How to Create Prototype Files	6-51
How to Specify Thunk Files	6-52
Deploy Applications That Use loadlibrary	6-52
loadlibrary in Parallel Computing Environment	6-52
Change Function Signature	6-52
Rename Library Function	6-52
Load Subset of Functions in Library	6-52
Call Function with Variable Number of Arguments	6-53

Intro to MEX-Files

7

Choosing MEX Applications	7-2
C++ MEX Functions	7-2

C/C++ MEX Functions for MATLAB R2017b and Earlier	7-2
Fortran MEX Functions	7-3
MEX Terms	7-3
MEX File Placement	7-4
MEX Files on Windows Network Drives	7-4
Use Help Files with MEX Functions	7-5
MATLAB Data	7-6
The MATLAB Array	7-6
Lifecycle of mxArray	7-6
Data Storage	7-7
MATLAB Data Types	7-9
Sparse Matrices	7-10
Using Data Types	7-11
Testing for Most-Derived Class	7-14
Testing for a Category of Types	7-14
Another Test for Built-In Types	7-15
Build C MEX Function	7-16
Change Default Compiler	7-18
Windows Systems	7-18
Use Non-Default Compiler on Mac Systems	7-19
Do Not Use mex -f optionsfile Syntax	7-19
Change Default gcc Compiler on Linux System	7-20
Custom Build with MEX Script Options	7-21
Include Files	7-21
Call LAPACK and BLAS Functions	7-22
Build matrixMultiply MEX Function Using BLAS Functions	7-22
Preserve Input Values from Modification	7-23
Pass Arguments to Fortran Functions from C/C++ Programs	7-23
Pass Arguments to Fortran Functions from Fortran Programs	7-24
Modify Function Name on UNIX Systems	7-25

Pass Separate Complex Numbers to Fortran Functions	7-27
Handling Complex Number Input Values	7-27
Handling Complex Number Output Arguments	7-27
Pass Complex Variables — matrixDivideComplex	7-28
Handle Fortran Complex Return Type — dotProductComplex	7-28
Symmetric Indefinite Factorization Using LAPACK — utdu_slv	7-29
MATLAB Support for Interleaved Complex API in MEX Functions	7-31
Separate Complex API and Interleaved Complex API	7-31
Matrix API Changes Supporting Interleaved Complex	7-32
Writing MEX Functions with Interleaved Complex API	7-34
MEX Functions Created in MATLAB R2017b and Earlier	7-34
Upgrade MEX Files to Use Interleaved Complex API	7-36
Check Array Complexity Using mxIsComplex	7-37
Add MX_HAS_INTERLEAVED_COMPLEX to Support Both Complex Number Representations	7-38
Use Typed Data Access Functions	7-39
Handle Complex mxArray Arrays	7-39
Maintain Complexity of mxArray	7-41
Replace Separate Complex Functions	7-42
Calculate Array Data Size with mxGetElementSize	7-42
Consider Replacing To-Be-Phased-Out Functions	7-42
Add Build Information to MEX Help File	7-43
Do I Need to Upgrade My MEX Files to Use Interleaved Complex API?	7-45
Can I Run Existing MEX Functions?	7-45
Must I Update My Source MEX Files?	7-45
Troubleshooting MEX API Incompatibilities	7-48
File Is Not A MEX File	7-48
MEX File Compiled With Incompatible Options	7-48
MEX File Compiled With One API And Linked With Another	7-48
C++ MEX File Using MATLAB Data API Compiled With Incompatible Option	7-48
Custom-built MEX File Not Supported In Current Release	7-49
MEX File Is Compiled With Outdated Option	7-49
MEX File Calls An Untyped Data Access Function	7-49
MEX File Calls A 32-bit Function	7-49

MEX File Does Not Contain An Entry Point	7-49
MEX File Built In MATLAB Release Not Supported In Current Release	7-49
Upgrade MEX Files to Use 64-Bit API	7-51
Back Up Files and Create Tests	7-52
Update Variables	7-52
Update Arguments Used to Call Functions in the 64-Bit API	7-52
Update Variables Used for Array Indices and Sizes	7-53
Analyze Other Variables	7-54
Replace Unsupported Functions	7-55
Test, Debug, and Resolve Differences After Each Refactoring Iteration	7-55
Resolve -largeArrayDims Build Failures and Warnings	7-55
Execute 64-Bit MEX File and Compare Results with 32-Bit Version	7-56
Experiment with Large Arrays	7-56
MATLAB Support for 64-Bit Indexing	7-58
What If I Do Not Upgrade?	7-59
Can I Run Existing Binary MEX Files?	7-59
Must I Update Source MEX Files on 64-Bit Platforms?	7-59
Upgrade Fortran MEX Files to use 64-bit API	7-62
Use Fortran API Header File	7-62
Declare Fortran Pointers	7-62
Require Fortran Type Declarations	7-62
Use Variables in Function Calls	7-63
Manage Reduced Fortran Compiler Warnings	7-64
Upgrade MEX Files to Use Graphics Objects	7-65
Replace mexGet and mexSet Functions	7-65
mex Automatically Converts Handle Type	7-67
I Want to Rebuild MEX Source Code Files	7-68
I Do Not Have MEX Source Code File	7-68
MEX Platform Compatibility	7-70
Invalid MEX File Errors	7-71
Run MEX File You Receive from Someone Else	7-73

MEX File Dependent Libraries	7-74
Document Build Information in the MEX File	7-75
MEX Version Compatibility	7-77
Getting Help When MEX Fails	7-78
Errors Finding Supported Compiler	7-78
Errors Building MEX Function	7-78
Preview mex Build Commands	7-79
MEX API Is Not Thread Safe	7-80
Compiling MEX File Fails	7-81
Build Example Files	7-81
Use Supported Compiler	7-81
File Not Found on Windows	7-81
Linux gcc -fPIC Errors	7-81
Compiler Errors in Fortran MEX Files	7-81
Syntax Errors Compiling C/C++ MEX Files on UNIX	7-82
Symbol mexFunction Unresolved or Not Defined	7-83
MEX File Segmentation Fault	7-84
MEX File Generates Incorrect Results	7-85
Memory Management Issues	7-86
Overview	7-86
Improperly Destroying an mxArray	7-87
Incorrectly Constructing a Cell or Structure mxArray	7-87
Creating a Temporary mxArray with Improper Data	7-88
Creating Potential Memory Leaks	7-88
Improperly Destroying a Structure	7-89
Destroying Memory in a C++ Class Destructor	7-90

Components of C MEX File	8-2
mexFunction Gateway Routine	8-2
Naming the MEX File	8-2
Required Parameters	8-2
Managing Input and Output Parameters	8-3
Validating Inputs	8-3
Computational Routine	8-4
MATLAB APIs	8-5
Matrix API	8-5
MEX Library	8-5
Preprocessor Macros	8-5
User Messages in C MEX Files	8-7
Error Handling in C MEX Files	8-8
Create C++ MEX Functions with C Matrix API	8-9
Creating Your C++ Source File	8-9
Compiling and Linking	8-9
Memory Considerations for Class Destructors	8-9
Use mexPrintf to Print to MATLAB Command Window	8-10
C++ Class Example	8-10
C++ File Handling Example	8-10
Create C Source MEX File	8-13
Tables of MEX Function Source Code Examples	8-20
Getting Started	8-20
C, C++, and Fortran MEX Functions	8-20
C MEX Functions Calling Fortran Subroutines	8-25
Choose a C++ Compiler	8-27
Select Microsoft Visual Studio Compiler	8-27
Select MinGW-w64 Compiler	8-27
Pass Scalar Values in C MEX File	8-29
Pass Scalar as Matrix	8-29
Pass Scalar by Value	8-30

Pass Strings in C MEX File	8-32
C Matrix API String Handling Functions	8-34
How MATLAB Represents Strings in MEX Files	8-34
Character Encoding and Multibyte Encoding Schemes	8-34
Converting MATLAB Character Vector to C-Style String	8-35
Converting C-Style String to MATLAB Character Vector	8-35
Returning Modified Input String	8-35
Memory Management	8-35
Pass Structures and Cell Arrays in C MEX File	8-37
Create 2-D Cell Array in C MEX File	8-39
Fill mxArray in C MEX File	8-41
Options	8-41
Copying Data Directly into an mxArray	8-41
Pointing to Data	8-41
Prompt User for Input in C MEX File	8-43
Handle Complex Data in C MEX File	8-44
Create Source File	8-44
Create Gateway Routine and Verify Input and Output Parameters	8-44
Create Output mxArray	8-45
Create Computational Routine	8-45
Call convec	8-46
Build and Test	8-46
Handle 8-, 16-, 32-, and 64-Bit Data in C MEX File	8-48
Manipulate Multidimensional Numerical Arrays in C MEX Files	8-49
Handle Sparse Arrays in C MEX File	8-51
Debug on Microsoft Windows Platforms	8-53
Notes on Debugging	8-54
Debug on Linux Platforms	8-55

Debug on Mac Platforms	8-57
Using Xcode	8-57
Using LLDB	8-59
Handling Large mxArray in C MEX Files	8-62
Using the 64-Bit API	8-62
Example	8-63
Caution Using Negative Values	8-64
Building Cross-Platform Applications	8-64
Typed Data Access in C MEX Files	8-65
Persistent mxArray	8-67
Automatic Cleanup of Temporary Arrays in MEX Files	8-69
Example	8-69
Handling Large File I/O in MEX Files	8-71
Prerequisites to Using 64-Bit I/O	8-71
Specifying Constant Literal Values	8-72
Opening a File	8-73
Printing Formatted Messages	8-74
Replacing fseek and ftell with 64-Bit Functions	8-74
Determining the Size of an Open File	8-75
Determining the Size of a Closed File	8-75
MinGW-w64 Compiler	8-77
Building yprime.c Example	8-77
MinGW Installation Folder Cannot Contain Space	8-77
Updating MEX Files to Use MinGW Compiler	8-78
Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64	8-79
Do Not Link to Library Files Compiled with Non-MinGW Compilers	8-79
MinGW Installation Folder Cannot Contain Space	8-79
MEX Command Does not Choose MinGW	8-79
Manually Configure MinGW for MATLAB	8-79
MinGW Behaves Similarly to gcc/g++ on Linux	8-80
Potential Memory Leak Inside C++ MEX Files on Using MEX Exceptions	8-80
Unhandled Explicit Exceptions in C++ MEX Files Unexpectedly Terminate MATLAB	8-81

C++ MEX Functions	9-2
C++ MEX API	9-2
Basic Design of C++ MEX Functions	9-2
Call MEX Function from MATLAB	9-3
Examples of C++ MEX Functions	9-3
Create a C++ MEX Source File	9-5
Create Source File	9-5
Add Required Header Files	9-5
Using Convenience Definitions	9-5
Define MexFunction Class	9-6
Define operator()	9-6
Add Member Function to Check Arguments	9-7
Implement Computation	9-8
Setup and Build	9-9
Call MEX Function	9-9
Build C++ MEX Programs	9-10
Supported Compilers	9-10
Build .cpp File with mex Command	9-10
MEX Include Files	9-10
File Extensions	9-11
Test Your Build Environment	9-12
C++ MEX API	9-14
matlab::mex::Function	9-14
matlab::mex::ArgumentList	9-14
C++ Engine API	9-15
matlab::engine::MATLABEngine::feval	9-16
matlab::engine::MATLABEngine::fevalAsync	9-18
matlab::engine::MATLABEngine::eval	9-20
matlab::engine::MATLABEngine::evalAsync	9-21
matlab::engine::MATLABEngine::getVariable	9-21
matlab::engine::MATLABEngine::getVariableAsync	9-22
matlab::engine::MATLABEngine::setVariable	9-23
matlab::engine::MATLABEngine::setVariableAsync	9-24
matlab::engine::MATLABEngine::getProperty	9-25
matlab::engine::MATLABEngine::getPropertyAsync	9-26
matlab::engine::MATLABEngine::setProperty	9-26

matlab::engine::MATLABEngine::setPropertyAsync	9-27
Exception Classes	9-28
Structure of C++ MEX Function	9-30
MEX Function Design	9-30
Header Files	9-30
Namespaces	9-30
Entry Point	9-30
Passing Arguments	9-31
Class Constructor and Destructor	9-32
Managing External Resources from MEX Functions	9-34
Reading a Text File	9-34
Handling Inputs and Outputs	9-37
Argument Validation	9-37
ArgumentList Iterators	9-38
Support Variable Number of Arguments	9-39
Data Access in Typed, Cell, and Structure Arrays	9-41
Shared Copies	9-41
Modify Cell Array in MEX Function	9-42
Modify Structure in MEX Function	9-44
Data Types for Passing MEX Function Data	9-48
Typed Arrays	9-48
Character Arrays	9-48
String Arrays	9-49
Cell Arrays	9-49
Structure Arrays	9-50
MATLAB Objects	9-50
Call MATLAB Functions from MEX Functions	9-52
Single Output	9-52
Multiple Outputs	9-53
Catch Exceptions in MEX Function	9-54
Execute MATLAB Statements from MEX Function	9-55
Set and Get MATLAB Variables from MEX	9-57
Get Variable from MATLAB Workspace	9-57
Put Variable in MATLAB Workspace	9-57

Set and Get Variable Example Code	9-58
MATLAB Objects in MEX Functions	9-60
Get Property Value	9-60
Get Property Value from Object Array	9-60
Set Property Value	9-61
Set Property Value in Object Array	9-61
Object Property Copy-On-Write Behavior	9-61
Modify Property and Return Object	9-61
Only Modified Property Causes Copy	9-63
Handle Objects	9-65
Avoid Copies of Arrays in MEX Functions	9-66
Input Array Not Modified	9-66
Input Array Modified	9-67
Displaying Output in MATLAB Command Window	9-69
Call MATLAB from Separate Threads in MEX Function	9-71
Communicating with MATLAB from Separate Threads	9-71
Example to Update the Display of Text	9-71
Out-of-Process Execution of C++ MEX Functions	9-75
How to Run Out of Process	9-75
Running arrayProduct MEX Function Out of Process	9-75
Process Life Cycle	9-76
Getting Information About the MEX Host Process	9-77
Always Run Out of Process	9-77
Debugging Out-of-Process MEX Functions	9-79
Making async Requests Using mexCallMATLAB	9-84

Fortran MEX-Files

10

Components of Fortran MEX File	10-2
mexFunction Gateway Routine	10-2
Naming the MEX File	10-2
Difference Between .f and .F Files	10-2
Required Parameters	10-3

Managing Input and Output Parameters	10-3
Validating Inputs	10-4
Computational Routine	10-4
MATLAB Fortran API Libraries	10-5
Matrix Library	10-5
MEX Library	10-5
Preprocessor Macros	10-5
Using the Fortran %val Construct	10-6
Data Flow in Fortran MEX Files	10-8
Showing Data Input and Output	10-8
Gateway Routine Data Flow Diagram	10-9
User Messages	10-11
Error Handling	10-12
Build Fortran MEX File	10-13
Create Fortran Source MEX File	10-14
Set Up Fortran Examples	10-19
Dynamically Allocate Memory	10-20
Call MATLAB Functions from Fortran MEX Files	10-21
Debug Fortran MEX Files	10-23
Notes on Debugging	10-23
Debugging on Microsoft Windows Platforms	10-23
Debugging on Linux Platforms	10-23
Handling Large mxArray Arrays	10-26
Using the 64-Bit API	10-26
Caution Using Negative Values	10-28
Building Cross-Platform Applications	10-28
Memory Management	10-29
MATLAB Supports Fortran 77	10-30

Handle Complex Fortran Data	10-31
Build and Test	10-32

Calling MATLAB Engine from C/C++ and Fortran Programs

11

Choosing Engine Applications	11-2
Introducing MATLAB Engine API for C and Fortran	11-4
Communicating with MATLAB Software	11-5
Callbacks in Applications	11-6
Call MATLAB Functions from C Applications	11-7
Call MATLAB Functions from Fortran Applications	11-9
Attach to Existing MATLAB Sessions	11-11
Build Windows Engine Application	11-13
Run Windows Engine Application	11-14
Set Run-Time Library Path on Windows Systems	11-15
Change Path Each Time You Run the Application	11-15
Permanently Change Path	11-15
Troubleshooting	11-16
Register MATLAB as a COM Server	11-17
Build Linux Engine Application	11-18
Build macOS Engine Application	11-19
Run Linux Engine Application	11-20
Run macOS Engine Application	11-21

Set Run-Time Library Path on Linux Systems	11-22
C Shell	11-22
Bourne Shell	11-22
Set Run-Time Library Path on macOS Systems	11-23
C Shell	11-23
Bourne Shell	11-23
Build Engine Applications with IDE	11-25
Configuring the IDE	11-25
Engine Include Files	11-25
Engine Libraries	11-26
Library Files Required by libeng	11-27
Can't Start MATLAB Engine	11-28
Multithreaded Applications	11-29
User Input Not Supported	11-30
Getting Started	11-31
Debug MATLAB Function Called by C Engine	11-32
Debug Engine Application on Windows	11-32
Debug Engine Application Attached to Existing MATLAB Session	11-32

Engine API for Java

12

MATLAB Engine API for Java	12-2
Build Java Engine Programs	12-3
General Requirements	12-3
Compile and Run Java Code on Windows	12-4
Compile and Run Java Code on macOS	12-4
Compile and Run Java Code on Linux	12-5
Java Example Source Code	12-7

Java Engine API Summary	12-8
com.mathworks Packages	12-8
com.mathworks.engine.MatlabEngine Methods	12-9
java.util.concurrent.Future Interface	12-10
Java Data Type Conversions	12-11
Pass Java Data to MATLAB	12-11
Pass MATLAB Data to Java	12-12
Start and Close MATLAB Session from Java	12-14
Start MATLAB Synchronously	12-14
Start MATLAB Asynchronously	12-14
Start Engine with Startup Options	12-15
Close MATLAB Engine Session	12-15
Connect Java to Running MATLAB Session	12-17
Find and Connect to MATLAB	12-17
Connect to MATLAB Synchronously	12-17
Connect to MATLAB Asynchronously	12-18
Specify Name of Shared Session	12-18
Execute MATLAB Functions from Java	12-20
Calling MATLAB Functions	12-20
Execute Function with Single Returned Argument	12-20
Execute Function with Multiple Returned Arguments	12-21
When to Specify Number of Output Arguments	12-21
Evaluate MATLAB Statements from Java	12-23
Evaluating MATLAB Statements	12-23
Evaluate Mathematical Function in MATLAB	12-23
Pass Variables from Java to MATLAB	12-25
Ways to Pass Variables	12-25
Pass Function Arguments	12-25
Put Variables in the MATLAB Workspace	12-26
Pass Variables from MATLAB to Java	12-27
Coordinate Conversion	12-27
Using Complex Variables in Java	12-28
Complex Variables in MATLAB	12-28
Get Complex Variables from MATLAB	12-28
Pass Complex Variable to MATLAB Function	12-29

Using MATLAB Structures in Java	12-31
MATLAB Structures	12-31
Pass Struct to MATLAB Function	12-31
Get Struct from MATLAB	12-32
Pass Java CellStr to MATLAB	12-34
MATLAB Cell Arrays	12-34
Create CellStr	12-34
Using MATLAB Handle Objects in Java	12-36
MATLAB Handle Objects	12-36
Java HandleObject Class	12-36
Set Graphics Object Properties from Java	12-36
Redirect MATLAB Command Window Output to Java	12-38
Output to MATLAB Command Window	12-38
Redirect MATLAB Output	12-38
Redirect MATLAB Error Messages to Java	12-39
Run Simulink Simulation from Java	12-40
MATLAB Command to Run Simulation	12-40
Run vdp Model from Java	12-40
MATLAB Engine API Exceptions	12-44
com.mathworks.engine.EngineException	12-44
com.mathworks.engine.UnsupportedTypeException	12-44
com.mathworks.engine.MatlabExecutionException	12-44
com.mathworks.engine.MatlabSyntaxException	12-44
Pass Java Array Arguments to MATLAB	12-46
2-D Array Arguments for MATLAB Functions	12-46
Multidimensional Arrays	12-46
Incorrect Java Data Types	12-48
Java String to MATLAB Character Vector	12-48
Setting Graphics Object Properties from Java	12-49
Java Integer to MATLAB double	12-49
Incorrect Number of Output Arguments	12-50

Get Started with MATLAB Engine API for Python	13-2
Install MATLAB Engine API for Python	13-5
Install MATLAB Engine API for Python in Nondefault Locations	13-7
Build or Install in Nondefault Folders	13-7
Install Engine in Your Home Folder	13-8
Start and Stop MATLAB Engine for Python	13-9
Start MATLAB Engine for Python	13-9
Run Multiple Engines	13-9
Stop Engine	13-9
Start Engine with Startup Options	13-10
Start Engine Asynchronously	13-10
Connect Python to Running MATLAB Session	13-11
Connect to Shared MATLAB Session	13-11
Connect Asynchronously to Shared MATLAB Session	13-12
Connect to Multiple Shared MATLAB Sessions	13-12
Start Shared MATLAB Sessions with Startup Options	13-13
Call MATLAB Functions from Python	13-14
Return Output Argument from MATLAB Function	13-14
Return Multiple Output Arguments from MATLAB Function	13-14
Return No Output Arguments from MATLAB Function	13-15
Stop Execution of Function	13-15
Call MATLAB Functions Asynchronously from Python	13-16
Call User Scripts and Functions from Python	13-17
Redirect Standard Output and Error to Python	13-19
Use MATLAB Handle Objects in Python	13-20
Use MATLAB Engine Workspace in Python	13-22

Pass Data to MATLAB from Python	13-23
Python Type to MATLAB Scalar Type Mapping	13-23
Python Container to MATLAB Array Type Mapping	13-24
Unsupported Python Types	13-24
Handle Data Returned from MATLAB to Python	13-25
MATLAB Scalar Type to Python Type Mapping	13-25
MATLAB Array Type to Python Type Mapping	13-26
Unsupported MATLAB Types	13-27
MATLAB Arrays as Python Variables	13-28
Create MATLAB Arrays in Python	13-28
MATLAB Array Attributes and Methods in Python	13-30
Multidimensional MATLAB Arrays in Python	13-30
Index Into MATLAB Arrays in Python	13-30
Slice MATLAB Arrays in Python	13-31
Reshape MATLAB Arrays in Python	13-32
Use MATLAB Arrays in Python	13-34
Sort and Plot MATLAB Data from Python	13-36
Get Help for MATLAB Functions from Python	13-40
How to Find MATLAB Help	13-40
Open MATLAB Help Browser from Python	13-40
Display MATLAB Help at Python Prompt	13-41
Default Numeric Types in MATLAB and Python	13-42
System Requirements for MATLAB Engine API for Python	13-44
Python Version Support	13-44
Download 64-Bit Versions of Python and MATLAB	13-44
Requirements for Building Python from Source	13-45
Limitations to MATLAB Engine API for Python	13-46
Troubleshoot MATLAB Errors in Python	13-47
MATLAB Errors in Python	13-47
MatlabExecutionError: Undefined Function	13-47
SyntaxError: Expression Not Valid Target	13-48
SyntaxError: Invalid Syntax	13-48
Cannot Find MATLAB Session in Python	13-49

Introduction to Engine API for C++	14-2
Getting Started	14-2
Basic Elements of C++ Engine Programs	14-2
C++ Engine API	14-5
Utility Functions	14-5
Classes	14-6
MATLABEngine Member Functions	14-6
Exception Classes	14-7
Data Size Limitations	14-8
Using Engine in Multiple-Thread Environment	14-8
Build C++ Engine Programs	14-9
Supported Compilers	14-9
Build .cpp File with mex Command	14-9
General Requirements	14-9
Engine Include Files	14-10
Runtime Environment	14-11
Test Your Build Environment	14-12
Start MATLAB Sessions from C++	14-14
Start MATLAB Session Synchronously	14-14
Start MATLAB Session Asynchronously	14-14
Start MATLAB with Startup Options	14-15
Connect C++ to Running MATLAB Session	14-16
Connect to Shared MATLAB	14-16
Connect to Shared MATLAB Asynchronously	14-17
Specify Name of Shared Session	14-17
Find and Connect to Named Shared Session	14-18
Call MATLAB Functions from C++	14-19
Call Function with Single Returned Argument	14-20
Call Function with Name/Value Arguments	14-22
Call Function Asynchronously	14-23
Call Function with Multiple Returned Arguments	14-23
Call Function with Native C++ Types	14-24
Control Number of Outputs	14-25

Evaluate MATLAB Statements from C++	14-29
Evaluation of MATLAB Statements	14-29
Evaluate Mathematical Function in MATLAB	14-29
Pass Variables from C++ to MATLAB	14-32
Ways to Pass Variables	14-32
Put Variables in MATLAB Base Workspace	14-32
Pass Variables from MATLAB to C++	14-34
Bring Result of MATLAB Calculation Into C++	14-34
Get MATLAB Objects and Access Properties	14-35
Set Property on MATLAB Object	14-35
Redirect MATLAB Command Window Output to C++	14-38
Redirect Screen Output	14-38
Redirect Error Output	14-39
Create Cell Arrays from C++	14-41
Put Cell Array in MATLAB Workspace	14-41
Access Elements of Cell Array	14-42
Modify Elements of Cell Array	14-42
Create Structure Arrays from C++	14-46
Create Structure Array and Send to MATLAB	14-46
Get Structure from MATLAB	14-47
Access Struct Array Data	14-47
Pass Enumerations to MATLAB From C++	14-50
Pass Sparse Arrays to MATLAB From C++	14-53
Run Simulink Simulation from C++	14-55
MATLAB Command to Run Simulation	14-55
Simulink vdp Model from C++	14-55
MATLAB Code to Run Simulation	14-56
C++ Code to Run Simulation	14-57
Convert C++ Engine Application to MATLAB Compiler SDK	
Application	14-59
Applications to Call Sierpinski Function	14-59

Read Cell Arrays of Excel Spreadsheet Data	15-4
Access a Simple .NET Class	15-6
System.DateTime Example	15-6
Create .NET Object from Constructor	15-7
View Information About .NET Object	15-7
Introduction to .NET Data Types	15-9
Load a Global .NET Assembly	15-12
Work with Microsoft Excel Spreadsheets Using .NET	15-13
Work with Microsoft Word Documents Using .NET	15-15
Assembly Is Library of .NET Classes	15-16
Limitations to .NET Support	15-17
System Requirements for Using MATLAB Interface to .NET	15-19
MATLAB Configuration File	15-19
Using .NET from MATLAB	15-20
Benefits of the MATLAB .NET Interface	15-20
Why Use the MATLAB .NET Interface?	15-20
.NET Assembly Integration Using MATLAB Compiler SDK ..	15-21
To Learn More About the .NET Framework	15-21
Using a .NET Object	15-22
Creating a .NET Object	15-22
What Classes Are in a .NET Assembly?	15-22
Using the delete Function on a .NET Object	15-23
Build a .NET Application for MATLAB Examples	15-24
Troubleshooting Security Policy Settings from Network Drives	15-25

.NET Terminology	15-26
.NET Framework System Namespace	15-26
Reference Type Versus Value Type	15-26
Simplify .NET Class Names	15-27
Use import in MATLAB Functions	15-28
Nested Classes	15-29
Handle .NET Exceptions	15-30
Pass Numeric Arguments	15-31
Call .NET Methods with Numeric Arguments	15-31
Use .NET Numeric Types in MATLAB	15-31
Pass System.String Arguments	15-32
Call .NET Methods with System.String Arguments	15-32
Use System.String in MATLAB	15-32
Pass System.Enum Arguments	15-34
Call .NET Methods with System.Enum Arguments	15-34
Use System.Enum in MATLAB	15-35
Pass System.Nullable Arguments	15-36
Pass Cell Arrays of .NET Data	15-40
Example of Cell Arrays of .NET Data	15-40
Create a Cell Array for Each System.Object	15-41
Create MATLAB Variables from the .NET Data	15-41
Call MATLAB Functions with MATLAB Variables	15-41
Pass Jagged Arrays	15-43
Create System.Double .NET Jagged Array	15-43
Call .NET Method with System.String Jagged Array Arguments	15-43
Call .NET Method with Multidimensional Jagged Array Arguments	15-45
Convert Nested System.Object Arrays	15-47
Pass Data to .NET Objects	15-49
Pass Primitive .NET Types	15-49

Pass Cell Arrays	15-50
Pass Nonprimitive .NET Objects	15-51
Pass MATLAB String and Character Data	15-51
Pass System.Nullable Type	15-52
Pass NULL Values	15-52
Unsupported MATLAB Types	15-52
Choosing Method Signatures	15-52
Example — Choosing a Method Signature	15-53
Pass Arrays	15-55
Pass MATLAB Arrays as Jagged Arrays	15-55
Handle Data Returned from .NET Objects	15-57
.NET Type to MATLAB Type Mapping	15-57
Convert Arrays of Primitive .NET Type to MATLAB Type	15-58
How MATLAB Handles System.String	15-59
How MATLAB Handles System._ComObject	15-59
How MATLAB Handles System.Nullable	15-61
How MATLAB Handles dynamic Type	15-61
How MATLAB Handles Jagged Arrays	15-62
Use Arrays with .NET Applications	15-63
Pass MATLAB Arrays to .NET	15-63
Convert Primitive .NET Arrays to MATLAB Arrays	15-63
Access .NET Array Elements in MATLAB	15-64
Convert .NET Jagged Arrays to MATLAB Arrays	15-65
Convert .NET Arrays to Cell Arrays	15-66
Convert Nested System.Object Arrays	15-66
cell Function Syntax for System.Object[,] Arrays	15-67
Limitations to Support of .NET Arrays	15-69
Set Static .NET Properties	15-70
Set System.Environment.CurrentDirectory Static Property	15-70
Do Not Use ClassName.PropertyName Syntax for Static Properties	15-71
Using .NET Properties	15-72
How MATLAB Represents .NET Properties	15-72
How MATLAB Maps C# Property and Field Access Modifiers	15-72

MATLAB Does Not Display Protected Properties	15-74
Work with .NET Methods Having Multiple Signatures	15-75
Display Function Signature Example	15-76
Call .NET Methods With out Keyword	15-77
Call .NET Methods With ref Keyword	15-79
Call .NET Methods With params Keyword	15-81
Call .NET Methods with Optional Arguments	15-83
Skip Optional Arguments	15-83
Call Overloaded Methods	15-84
Calling .NET Methods	15-86
Getting Method Information	15-86
C# Method Access Modifiers	15-86
VB.NET Method Access Modifiers	15-87
Reading Method Signatures	15-87
Calling .NET Methods with Optional Arguments	15-88
Skipping Optional Arguments	15-88
Determining Which Overloaded Method Is Invoked	15-88
Support for ByRef Attribute in VB.NET	15-88
Calling .NET Extension Methods	15-89
Call .NET Properties That Take an Argument	15-90
How MATLAB Represents .NET Operators	15-92
Limitations to Support of .NET Methods	15-93
Displaying Generic Methods	15-93
Overloading MATLAB Functions	15-93
Calling Overloaded Static Methods	15-93
Use .NET Events in MATLAB	15-94
Monitor Changes to .TXT File	15-94
Monitor Changes to Windows Form ComboBox	15-94
Call .NET Delegates in MATLAB	15-97
Declare a Delegate in a C# Assembly	15-97

Load the Assembly Containing the Delegate into MATLAB .	15-97
Select a MATLAB Function	15-97
Create an Instance of the Delegate in MATLAB	15-97
Invoke the Delegate Instance in MATLAB	15-98
Create Delegates from .NET Object Methods	15-99
Create Delegate Instances Bound to .NET Methods	15-101
Example — Create a Delegate Instance Associated with a .NET Object Instance Method	15-101
Example — Create a Delegate Instance Associated with a Static .NET Method	15-102
.NET Delegates With out and ref Type Arguments	15-103
Combine and Remove .NET Delegates	15-104
.NET Delegates	15-106
Calling .NET Methods Asynchronously	15-107
How MATLAB Handles Asynchronous Method Calls in .NET	15-107
Using EndInvoke With out and ref Type Arguments	15-107
Using Polling to Detect When Asynchronous Call Finishes	15-107
Limitations to Support of .NET Events	15-108
MATLAB Support of Standard Signature of an Event Handler Delegate	15-108
Limitations to Support of .NET Delegates	15-109
Use Bit Flags with .NET Enumerations	15-110
How MATLAB Supports Bitwise Operations on System.Enum	15-110
Creating .NET Enumeration Bit Flags	15-110
Removing a Flag from a Variable	15-111
Replacing a Flag in a Variable	15-112
Testing for Membership	15-112
Read Special System Folder Path	15-114
Default Methods for an Enumeration	15-116

NetDocEnum Example Assembly	15-118
Work with Members of a .NET Enumeration	15-119
Refer to a .NET Enumeration Member	15-121
Using the Implicit Constructor	15-121
Display .NET Enumeration Members as Character Vectors	15-123
Convert .NET Enumeration Values to Type Double	15-124
Iterate Through a .NET Enumeration	15-125
Information About System.Enum Methods	15-125
Display Enumeration Member Names	15-126
Use .NET Enumerations to Test for Conditions	15-127
Using Switch Statements	15-127
Using Relational Operations	15-127
Underlying Enumeration Values	15-129
Limitations to Support of .NET Enumerations	15-130
Create .NET Collections	15-131
Convert .NET Collections to MATLAB Arrays	15-133
Create .NET Arrays of Generic Type	15-134
Display .NET Generic Methods Using Reflection	15-135
showGenericMethods Function	15-135
Display Generic Methods in a Class	15-136
Display Generic Methods in a Generic Class	15-137
.NET Generic Classes	15-138
Accessing Items in .NET Collections	15-139
Call .NET Generic Methods	15-140
Using the NetDocGeneric Example	15-140
Invoke Generic Class Member Function	15-140
Invoke Static Generic Functions	15-141

Invoke Static Generic Functions of a Generic Class	15-141
Invoke Generic Functions of a Generic Class	15-142

Using COM Objects from MATLAB

16

MATLAB COM Integration	16-2
Concepts and Terminology	16-2
COM Objects, Clients, and Servers	16-2
Interfaces	16-2
MATLAB COM Client	16-3
MATLAB COM Automation Server	16-3
 Register Servers	 16-4
Access COM Controls Created with .NET	16-4
Verify Registration	16-4
 Get Started with COM	 16-5
Create Instance of COM Object	16-5
Register Custom Control	16-5
 Read Spreadsheet Data Using Excel as Automation Server	 16-6
Techniques Demonstrated	16-6
Create Excel Automation Server	16-6
Manipulate Data in MATLAB Workspace	16-7
Create Plotter Interface	16-8
Insert MATLAB Graphs into Excel Spreadsheet	16-10
Run Example	16-11
 Supported Client/Server Configurations	 16-13
MATLAB Client and In-Process Server	16-13
MATLAB Client and Out-of-Process Server	16-13
COM Implementations Supported by MATLAB	16-14
Client Application and MATLAB Automation Server	16-14
Client Application and MATLAB Engine Server	16-15

Create COM Objects	17-2
Instantiate DLL Component	17-2
Instantiate EXE Component	17-2
Handle COM Data in MATLAB	17-4
Pass Data to COM Objects	17-4
Handle Data from COM Objects	17-5
Unsupported Types	17-7
Pass MATLAB SAFEARRAY to COM Object	17-7
Read SAFEARRAY from COM Objects in MATLAB Applications	17-9
Display MATLAB Syntax for COM Objects	17-9
COM Object Properties	17-12
MATLAB Functions for Object Properties	17-12
Work with Multiple Objects	17-12
Enumerated Values for Properties	17-12
Property Inspector	17-13
Custom Properties	17-14
Properties That Take Arguments	17-14
COM Methods	17-15
Method Information	17-15
Call Object Methods	17-15
Specify Enumerated Parameters	17-16
Skip Optional Input Arguments	17-16
Return Multiple Output Arguments	17-17
COM Events	17-18
COM Event Handlers	17-20
Arguments Passed to Event Handlers	17-20
Event Structure	17-21
COM Object Interfaces	17-22
IUnknown and IDispatch Interfaces	17-22
Save and Delete COM Objects	17-24
Functions to Save and Restore COM Objects	17-24

Release COM Interfaces and Objects	17-24
MATLAB Application as DCOM Client	17-26
Explore COM Objects	17-27
Properties	17-27
Methods	17-27
Events	17-28
Interfaces	17-28
Identify Objects and Interfaces	17-29
Change Row Height in Range of Spreadsheet Cells	17-31
Write Data to Excel Spreadsheet	17-34
Change Cursor in Spreadsheet	17-36
Insert Spreadsheet After First Sheet	17-38
Connect to Existing Excel Application	17-40
Display Message for Workbook OnClose Event	17-41
COM Collections	17-42
MATLAB COM Support Limitations	17-43
Interpreting Argument Callouts in COM Error Messages ..	17-44

MATLAB COM Automation Server Support

18

Register MATLAB as Automation Server	18-2
When to Register MATLAB	18-2
Register from System Prompt	18-2
Register from MATLAB Command Prompt	18-3
Unregister MATLAB as Automation Server	18-3
MATLAB COM Automation Server Interface	18-4
COM Server Types	18-4

Programmatic Identifiers	18-4
Shared and Dedicated Servers	18-5
In-Process and Out-of-Process Servers	18-5
Create MATLAB Server	18-7
Getting Started	18-7
Get or Set the Status of a MATLAB Automation Server	18-8
Connect to Existing MATLAB Server	18-9
Using Visual Basic .NET Code	18-9
VT_DATE Data Type	18-10
Data Types For Visual Basic .NET Clients	18-11
Visible Property	18-12
Manually Create Automation Server	18-13
Start MATLAB as Automation Server in Desktop Mode	18-14
Call MATLAB Function from Visual Basic .NET Client	18-15
Pass Complex Data to MATLAB from C# Client	18-16
Call MATLAB Function from C# Client	18-18
View MATLAB Functions from Visual Basic .NET Object Browser	18-20
Waiting for MATLAB Application to Complete	18-21
Conversion of MATLAB Types to COM Types	18-22
Variant Data	18-22
SAFEARRAY Data	18-23
Unsupported Types	18-23
Conversion of COM Types to MATLAB Types	18-24

Display Streamed Data in Figure Window	19-2
PricesStreamer Class	19-2
Map Data to MATLABuitable Object	19-5
Display Data in JSON Format	19-6
Terminate Data Stream	19-6
Call PricesStreamer	19-7
Display JPEG Images Streamed from IP Camera	19-8
CameraPlayer Class	19-8
Call CameraPlayer	19-10
Send Multipart Form Messages	19-12
Send and Receive HTTP Messages	19-14
What Is the HTTP Interface?	19-18
Manage Cookies	19-19
HTTP Data Type Conversion	19-21
Converting Data in Request Messages	19-21
Converting Data in Response Messages	19-26
Supported Image Data Subtypes	19-28
Display Progress Monitor for HTTP Message	19-30
Set Up WSDL Tools	19-34
Display a World Map	19-35
Using WSDL Web Service with MATLAB	19-40
What Are Web Services in MATLAB?	19-40
What Are WSDL Documents?	19-40
What You Need to Use WSDL with MATLAB	19-41
Access Services That Use WSDL Documents	19-43
Error Handling	19-45
Considerations Using Web Services	19-45

Error Handling with try/catch Statements	19-45
Use a Local Copy of the WSDL Document	19-45
Java Errors Accessing Service	19-46
Anonymous Types Not Supported	19-46
XML-MATLAB Data Type Conversion	19-47
Limitations to WSDL Document Support	19-49
Unsupported WSDL Documents	19-49
Documents Must Conform to Wrapper Style	19-51
SOAP Header Fields Not Supported	19-52
Manually Redirect HTTP Messages	19-53

Python Interface Topics

20

Install Supported Python Implementation	20-3
Call Python from MATLAB	20-4
Call User-Defined Python Module	20-7
Use Python Numeric Types in MATLAB	20-9
Call Python Methods with Numeric Arguments	20-10
Default Numeric Types	20-11
Use Python array Types in MATLAB	20-12
Pass MATLAB Character Vector to Python Method	20-13
Use Python str Type in MATLAB	20-14
Pass MATLAB Backslash Control Character	20-16
Create Python list Variable	20-17

Use Python list Type in MATLAB	20-18
Use Python List of Numeric Types in MATLAB	20-20
Pass Cell Array as Python Sequence Type	20-22
Read Element of Nested list Type	20-24
Create Python tuple Variable	20-25
Create Singleton Python tuple Variable	20-26
Create Python dict Variable	20-27
Pass dict Argument to Python Method	20-28
Use Python dict Type in MATLAB	20-29
Convert Python dict Type to MATLAB Structure	20-31
Pass Keyword Arguments	20-32
Pass Python Function to Python map Function	20-33
Index into Python String	20-34
Index into Python List	20-35
Index into Python Tuple	20-37
Index into Python dict	20-38
Use Python List as Values in for Loop	20-40
Display Stepped Range of Python Elements	20-41
Access Elements in Python Container Types	20-42
Sequence Types	20-42
Mapping Types	20-43
Size and Dimensions	20-43
Array Support	20-44
Use Zero-Based Indexing for Python Functions	20-44

View Python Numeric Values	20-45
Why Do I See Properties When I Display a Number?	20-45
What Is the L Character Attached to a Number?	20-45
Call Methods on Python Variables	20-47
Reload Modified User-Defined Python Module	20-49
System and Configuration Requirements	20-51
Python Version Support	20-51
Set Python Version on Windows Platform	20-51
Set Python Version on Mac and Linux Platforms	20-52
Download 64-Bit Version of Python on Windows Platforms .	20-52
Requirements for Building Python Executable	20-52
Create a Python Object	20-53
Pass Data to Python	20-56
MATLAB Type to Python Type Mapping	20-56
MATLAB Vector to Python Mapping	20-57
Passing Matrices and Multidimensional Arrays	20-57
Troubleshooting Multidimensional Array Argument Errors	20-61
Unsupported MATLAB Types	20-62
Handle Data Returned from Python	20-63
Automatic Python Type to MATLAB Type Mapping	20-63
Explicit Type Conversions	20-63
How MATLAB Represents Python Operators	20-66
Execute Callable Python Object	20-68
Understanding Python and MATLAB import Commands ...	20-69
Do Not Type import pythonmodule	20-69
Use MATLAB import to Shorten Class or Function Names ..	20-69
List, Tuple, and Dictionary Types	20-71
Limitations to Python Support	20-72

Limitations to Indexing into Python Objects	20-74
Undefined variable "py" or function "py.command"	20-75
Python Not Installed	20-75
64-bit/32-bit Versions of Python on Windows Platforms	20-75
MATLAB Cannot Find Python	20-75
Error in User-Defined Python Module	20-76
Python Module Not on Python Search Path	20-76
Module Name Conflicts	20-76
Python Tries to Execute command in Wrong Module	20-77
Help for Python Functions	20-78
Handle Python Exceptions	20-80
Troubleshooting Error Messages	20-81
Python Error: Python class: message	20-81
Python Module Errors	20-81
Errors Converting Python Data	20-82
Using Python Data in MATLAB	20-83
Call Python eval Function	20-84
Precedence Order of Methods and Functions	20-86
Python Function Arguments	20-87
Positional Arguments	20-87
Keyword Arguments	20-87
Optional Arguments	20-88
Out-of-Process Execution of Python Functionality	20-90
Limitations	20-91

System Commands

21

Shell Escape Function Example	21-2
--	-------------

Run External Commands, Scripts, and Programs	21-3
Shell Escape Function	21-3
Return Results and Status	21-3
Specify Environment Variables	21-4
Run UNIX Programs off System Path	21-4
Run AppleScript on macOS	21-5
 Change Environment Variable for Shell Command	 21-7

Settings

22

Access and Modify Settings	22-2
Access Settings	22-2
Modify Settings	22-3
Restore Default Values	22-4
Settings and Preferences	22-4
 Create Custom Settings	 22-6
Add and Remove Settings Groups	22-6
Add and Remove Settings	22-7
Validate Settings Using Functions	22-8
 Create Factory Settings for Toolboxes	 22-13
Create Factory Settings Tree	22-13
Create Factory Settings JSON File	22-18
Test Factory Settings Tree	22-18
Ensure Backward Compatibility Across Toolbox Versions ..	22-19

Serial Port I/O

23

Serial Port Overview	23-2
What Is Serial Communication?	23-2
Serial Port Interface Standard	23-2
Supported Platforms	23-3
Connecting Two Devices with a Serial Cable	23-3

Serial Port Signals and Pin Assignments	23-4
Serial Data Format	23-8
Find Serial Port Information for Your Platform	23-11
Create Serial Port Object	23-16
Create a Serial Port Object	23-16
Serial Port Object Display	23-17
Configure Serial Port Communication Settings	23-19
Write and Read Serial Port Data	23-21
Rules for Completing Write and Read Operations	23-21
Writing and Reading Text Data	23-21
Writing and Reading Binary Data	23-23
Use Callbacks for Serial Port Communication	23-26
Callback Properties	23-26
Using Callbacks	23-26
Use Serial Port Control Pins	23-28
Control Pins	23-28
Signaling the Presence of Connected Devices	23-28
Controlling the Flow of Data: Handshaking	23-31
Getting Started with Serial Port Communication	23-34
Query a Serial Port Device	23-34
The Serial Port Session	23-34
Configuring and Returning Properties	23-35
Transition Your Code to serialport Interface	23-40
Discover Serial Port Devices	23-40
Connect to Serial Port Device	23-41
Read and Write	23-41
Send a Command	23-42
Read a Terminated String	23-43
Flush Data from Memory	23-44
Set Terminator	23-44
Set Up a Callback Function	23-45
Read Serial Pin Status	23-46
Set Serial DTR and RTS Pin States	23-46
Read Streaming Data from Arduino Using Serial Port Communication	23-47

Capabilities and Supported Interfaces and Platforms	23-50
What Is the MATLAB Serial Port Interface?	23-50
Supported Serial Port Interface Standards	23-51
Supported Platforms	23-51
Using the Examples with Your Device	23-51
Find Available Serial Ports Using the serialist Function . . .	23-51
Overview of the Serial Port	23-53
Introduction	23-53
What Is Serial Communication?	23-53
The Serial Port Interface Standard	23-53
Connecting Two Devices with a Serial Cable	23-54
Serial Port Signals and Pin Assignments	23-55
Serial Data Format	23-59
Finding Serial Port Information for Your Platform	23-62
Using Virtual USB Serial Ports	23-65
Selected Bibliography	23-65
Create a Serial Port Object	23-67
Overview of a Serial Port Object	23-67
Configuring Properties During Object Creation	23-69
The Serial Port Object Display	23-69
Creating an Array of Serial Port Objects	23-70
Connect to the Device	23-71
Configure Communication Settings	23-72
Write and Read Data	23-73
Before You Begin	23-73
Example — Introduction to Writing and Reading Data	23-73
Controlling Access to the MATLAB Command Line	23-74
Writing Data	23-75
Reading Data	23-80
Example — Writing and Reading Text Data	23-86
Example — Parsing Input Data Using textscan	23-87
Example — Reading Binary Data	23-88
Events and Callbacks	23-91
Introduction	23-91
Example — Introduction to Events and Callbacks	23-91
Event Types and Callback Properties	23-92
Respond To Event Information	23-94

Create and Execute Callback Functions	23-96
Enable Callback Functions After They Error	23-97
Example — Using Events and Callbacks	23-97
Control Pins	23-99
Properties of Serial Port Control Pins	23-99
Signaling the Presence of Connected Devices	23-99
Controlling the Flow of Data: Handshaking	23-102
Debugging: Recording Information to Disk	23-105
Introduction	23-105
Recording Properties	23-105
Example: Introduction to Recording Information	23-106
Creating Multiple Record Files	23-106
Specifying a Filename	23-107
The Record File Format	23-107
Example: Recording Information to Disk	23-108
Save and Load	23-111
Using save and load	23-111
Using Serial Port Objects on Different Platforms	23-111
Disconnect and Clean Up	23-113
Disconnect a Serial Port Object	23-113
Clean Up the MATLAB Environment	23-113
Property Reference	23-115
Configure Serial Port Properties	23-115
Serial Port Object Properties	23-115
Properties — Alphabetical List	23-118

Hardware Support

24

Support Package Installation	24-2
What Is a Support Package?	24-2
Install Support Packages	24-2
Change Your Support Package Directory	24-4

External Programming Languages and Systems

Integrate MATLAB with External Programming Languages and Systems

MATLAB provides a flexible, two-way integration with other programming languages, allowing you to reuse legacy code. For a list of programming languages and the supported versions, see MATLAB Supported Interfaces to Other Languages.

Call C/C++ Code from MATLAB

MATLAB provides these features to help you integrate C/C++ algorithms into your MATLAB applications.

- A C/C++ shared library interface is a collection of functions dynamically loaded by an application at run time. Using a shared library has the advantage of packaging multiple library functions into one interface. In addition, MATLAB manages data type conversions.
 - Call C++ Library Functions - To call functions in a C++ shared library, use the `clib` package described in “C++ Libraries”.
 - Whenever possible, choose the C++ interface over the C-only interface. For information about C++ support, see these limitations on page 5-44.
 - To call functions in a C shared library, use the `calllib` function. For information, see “C Libraries”. This feature works best with C-only libraries, but has these limitations on page 6-7.

If you want more control over data conversion and memory management, consider writing a MEX file.

- A MEX file is wrapper code around a C/C++ algorithm that handles the conversion of MATLAB data types to C types. MEX files provide better performance than calling functions through MATLAB shared library interfaces. Also, MEX files give you more programmatic control over data conversion and memory management.
 - “C++ MEX Applications” use modern C++ programming features and, where possible, shared copies of data.
 - “C MEX File Applications” use the “C Matrix API” and is supported for existing MEX functions. MathWorks recommends that whenever possible, choose C++ MEX over C MEX file applications. However, if your MEX functions must run in MATLAB R2017b or earlier, then write MEX functions with the C matrix library.

- If you have multiple functions in a library or do not have performance issues, consider writing a C++ library interface.

These features require C/C++ programming skills to create a library interface or to write a MEX function. However, you can give the resulting library or MEX function to any MATLAB user. The end user calls the functionality like any MATLAB function, without knowing the underlying details of the C/C++ language implementation.

To call MATLAB from a C/C++ language program, see “MATLAB Engine API for C++” or “MATLAB Engine API for C”.

Use Objects from Other Programming Languages in MATLAB

If you have functions and objects in another programming language, you can call them from MATLAB. You do not need to be a software developer to integrate these objects into your MATLAB application. However, you need access to third-party documentation for the libraries.

MATLAB supports calling functions and using objects in the following languages.

- “C++ Libraries”
- “C Libraries”
- MEX File Functions for C/C++ and Fortran
- “Java Libraries”
- “Python Libraries”
- “.NET Libraries”
- COM Objects

Call MATLAB from Another Programming Language

You can call MATLAB from another language using Engine Applications on page 11-2. Using MATLAB engine APIs, call MATLAB functions from your own application. MATLAB has APIs for the following languages.

- Engine API for C++
- Engine API for Java
- Engine API for Python

- Engine API for C
- Engine API for Fortran

To create an engine application, install a MATLAB-supported compiler and use the `mex` command to build the application.

Call Your Functions as MATLAB Functions

You can write your own functions and call them as MATLAB functions using MEX APIs. For more information, see “Choosing MEX Applications” on page 7-2. You can write MEX functions in the following languages.

- C++ MEX APIs
- C MEX APIs
- Fortran MEX APIs

To create a MEX file, install a MATLAB-supported compiler and use the `mex` command to build the function.

Communicate with Web Services

You can communicate with web services from MATLAB.

- MATLAB RESTful web services functions allow non-programmers to access many web services using HTTP GET and POST methods.
- For functionality not supported by the RESTful web services functions, use the HTTP Interface classes for writing customized web access applications.
- If your web service is based on Web Services Description Language (WSDL) document technologies, then use the MATLAB WSDL functions.

See Also

More About

- “External Language Interfaces”

External Websites

- [MATLAB Supported Interfaces to Other Languages](#)
- [Supported and Compatible Compilers](#)

Custom Linking to Required API Libraries

MathWorks recommends that you use the `mex` command to build MEX files and engine applications. This build script automatically links to the libraries required by the MATLAB APIs used in your application.

To custom build these applications using an Integrated Development Environment (IDE) instead of the `mex` command, refer to this list of required run-time libraries and include files. To identify path names, use these MATLAB commands.

- Replace `matlabroot` with the value returned by `matlabroot`.
- Replace `compiler` with either `microsoft` or `mingw64`.
- The path to the include files is the value returned by:

```
fullfile(matlabroot, 'extern', 'include')
```

C++ MEX Functions

To build C++ MEX functions, use the “C++ MEX API” on page 9-14 and “MATLAB Data API”.

Include files:

- `mex.hpp` — Definitions for the C++ MEX API
- `mexAdapter.hpp` — Utilities required by the C++ MEX function operator

Windows® libraries:

- `matlabroot\extern\lib\win64\compiler\libMatlabDataArray.lib`

Linux® libraries:

- `Linux—matlabroot/extern/bin/glnxa64/libMatlabDataArray.so`

macOS libraries:

- `macOS—matlabroot/extern/bin/maci64/libMatlabDataArray.dylib`

C++ Engine Applications

To build C++ engine applications, use the “MATLAB Engine API for C++” and “MATLAB Data API”.

Include files:

- `MatlabEngine.hpp` — Definitions for the C++ engine API
- `MatlabDataArray.hpp` — Definitions for MATLAB Data Arrays

Windows libraries:

- Engine library — `matlabroot\extern\lib\win64\compiler\libMatlabEngine.lib`
- MATLAB Data Array library — `matlabroot\extern\lib\win64\compiler\libMatlabDataArray.lib`

Linux libraries:

- Engine library — `matlabroot/extern/bin/glnxa64/libMatlabEngine.so`
- MATLAB Data Array library — `matlabroot/extern/bin/glnxa64/libMatlabDataArray.so`

macOS libraries:

- Engine library — `matlabroot/extern/bin/maci64/libMatlabEngine.dylib`
- MATLAB Data Array library — `matlabroot/extern/bin/maci64/libMatlabDataArray.dylib`

C MEX Functions

To build C MEX functions, use the “C Matrix API” and C MEX API functions listed in “C MEX File Applications”. Optionally, to read or write to MAT-files in your MEX functions, use the “MATLAB C API to Read MAT-File Data”.

Include files:

- `mex.h` — Declares the entry point and interface routines
- `matrix.h` — Definitions of the `mxArray` structure and function prototypes for matrix access routines

- `mat.h` (optional) — Function prototypes for `mat` routines

Windows libraries:

- `matlabroot\extern\lib\win64\compiler\libmex.lib`
- `matlabroot\extern\lib\win64\compiler\libmx.lib`
- `matlabroot\extern\lib\win64\compiler\libmat.lib` (optional)

Linux libraries:

- `matlabroot/bin/glnxa64/libmex.so`
- `matlabroot/bin/glnxa64/libmx.so`
- `matlabroot/bin/glnxa64/libmat.so` (optional)

macOS libraries:

- `matlabroot/bin/maci64/libmex.dylib`
- `matlabroot/bin/maci64/libmx.dylib`
- `matlabroot/bin/maci64/libmat.dylib` (optional)

C Engine Applications

To build C engine applications, use the “C Matrix API” and “MATLAB Engine API for C”. If you include C MEX API functions such as `mexPrintf` in your application, then you must link to the `libmex` library. For a list of functions, see “C MEX File Applications”. Optionally, to read or write MAT-files in your application, use the “MATLAB C API to Read MAT-File Data”.

Include files:

- `engine.h` — Function prototypes for engine routines
- `matrix.h` — Definition of the `mxArray` structure and function prototypes for matrix access routines
- `mat.h` (optional) — Function prototypes for `mat` routines

Windows libraries:

- Engine library — `matlabroot\extern\lib\win64\compiler\libeng.lib`

- Matrix library — *matlabroot\extern\lib\win64\compiler\libmx.lib*
- MEX library (optional) — *matlabroot\extern\lib\win64\compiler\libmex.lib*
- MAT-File library (optional) — *matlabroot\extern\lib\win64\compiler\libmat.lib*

Linux libraries:

- Engine library — *matlabroot/bin/glnxa64/libeng.so*
- Matrix library — *matlabroot/bin/glnxa64/libmx.so*
- MEX library (optional) — *matlabroot/bin/glnxa64/libmex.so*
- MAT-File library (optional) — *matlabroot/bin/glnxa64/libmat.so*

macOS libraries:

- Engine library — *matlabroot/bin/maci64/libeng.dylib*
- Matrix library — *matlabroot/bin/maci64/libmx.dylib*
- MEX library (optional) — *matlabroot/bin/maci64/libmex.dylib*
- MAT-File library (optional) — *matlabroot/bin/maci64/libmat.dylib*

C MAT-File Applications

To build standalone applications to read data from C MAT-files, use the “C Matrix API” and “MATLAB C API to Read MAT-File Data”. If you include C MEX API functions such as `mexPrintf` in your application, then you must link to the `libmex` library. For a list of these functions, see “C MEX File Applications”.

Include files:

- `mat.h` — Function prototypes for `mat` routines
- `matrix.h` — Definitions of the `mxArray` structure and function prototypes for matrix access routines

Windows libraries:

- MAT-File library — *matlabroot\extern\lib\win64\compiler\libmat.lib*
- Matrix library — *matlabroot\extern\lib\win64\compiler\libmx.lib*

- MEX library (optional) — *matlabroot\extern\lib\win64\compiler\libmex.lib*

Linux libraries:

- MAT-File library — *matlabroot/bin/glnxa64/libmat.so*
- Matrix library — *matlabroot/bin/glnxa64/libmx.so*
- MEX library (optional) — *matlabroot/extern/bin/glnxa64/libmex.so*

macOS libraries:

- MAT-File library — *matlabroot/bin/maci64/libmat.dylib*
- Matrix library — *matlabroot/bin/maci64/libmx.dylib*
- MEX library (optional) — *matlabroot/extern/bin/maci64/libmex.dylib*

See Also

mex

More About

- “MEX File Dependent Libraries” on page 7-74
- “Build C++ Engine Programs” on page 14-9
- “Build Engine Applications with IDE” on page 11-25
- “MAT-File API Library and Include Files” on page 4-7

External Websites

- Compiling MEX Files without the mex Command

External Data Interface (EDI)

- “Create Arrays with C++ MATLAB Data API” on page 2-2
- “Copy C++ MATLAB Data Arrays” on page 2-5
- “C++ Cell Arrays” on page 2-7
- “Access C++ Data Array Container Elements” on page 2-9
- “Operate on C++ Arrays Using Visitor Pattern” on page 2-11
- “MATLAB Data API Exceptions” on page 2-17
- “MATLAB Data API Types” on page 2-21

Create Arrays with C++ MATLAB Data API

In this section...

“Create Arrays” on page 2-2

“Operate on Each Element in an Array” on page 2-3

Create Arrays

The C++ MATLAB Data API lets applications running outside of MATLAB work with MATLAB data through a MATLAB-neutral interface. The API uses modern C++ semantics and design patterns and avoids data copies whenever possible by using MATLAB copy-on-write semantics.

The header file for the MATLAB Data API is `MatlabDataArray.hpp`.

The `matlab::data::Array` class is the base class for all array types. It provides general array information, such as type and size. The `Array` class supports both one-dimensional and multi-dimensional arrays. The MATLAB Data API uses zero-based indexing.

To create an array, first create a factory using `matlab::data::ArrayFactory`.

```
matlab::data::ArrayFactory factory;
```

Use the factory to create a 2-by-2 array of type `double`. Specify the array values in column-major format to match the ordering of the MATLAB statement `A = [1 2; 3 4]`. To inspect the array, use the functions in the `matlab::data::Array` class.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    Array A = factory.createArray<double>({ 2,2 },
        { 1.0, 3.0, 2.0, 4.0 });

    // Inspect array
    ArrayType c = A.getType();
    ArrayDimensions d = A.getDimensions();
    size_t n = A.getNumberOfElements();
}
```

```
    return 0;
}
```

This code is equivalent to the following MATLAB statements.

```
A = [1 2; 3 4];
c = class(A);
d = size(A);
n = numel(A);
```

The `matlab::data::TypedArray` class supports iterators, which enable you to use range-based for loops. This example creates a 1-by-6 array from the 3-by-2 `TypedArray`.

```
#include "MatlabDataArray.hpp"

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

    // Create a 3-by-2 TypedArray
    TypedArray<double> A = factory.createArray( {3,2},
        {1.1, 2.2, 3.3, 4.4, 5.5, 6.6 });

    // Assign values of A to the double array C
    double C[6];
    int i = 0;
    for (auto e : A) {
        C[i++] = e;
    }

    return 0;
}
```

Operate on Each Element in an Array

Modify each element in a `matlab::data::Array` using a reference to the element. This example multiplies each element in the `matlab::data::TypedArray` by a scalar value.

```
#include "MatlabDataArray.hpp"

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

```
// Create a 3-by-2 TypedArray
TypedArray<double> A = factory.createArray( {3,2},
    {1.1, 2.2, 3.3, 4.4, 5.5, 6.6 });

// Define scalar multiplier
double multiplier(10.2);

// Multiple each element in A
for (auto& elem : A) {
    elem *= multiplier;
}

return 0;
}
```

See Also

`createArray` | `matlab::data::TypedArray`

Copy C++ MATLAB Data Arrays

The `matlab::data::Array` class supports both copy and move semantics. Copies of `Array` objects create shared data copies. In the following C++ code, variables `B` and `C` are copies of `matlab::data::CharArray` `A`; all three variables point to the same data.

```
#include "MatlabDataArray.hpp"

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

    // Create a shared copy of A
    CharArray B(A);

    CharArray C = factory.createCharArray("");
    // Copy the contents of A into C
    C = A;

    return 0;
}
```

`Array` supports copy-on-write semantics. Copies of an `Array` object are unshared when a write operation is performed. In the previous example, modifying the variable `B` creates a copy of the `CharArray` object with updated data. However, `A` and `C` remain shared copies.

```
// B becomes unshared once modified
B[20] = char16_t(33);
```

C++ MATLAB Data Arrays support move semantics. When you pass a variable using `move`, there is no copy of the variable.

Avoid Unnecessary Data Copying

If you index into or use an iterator on an array for read-only purposes, then the best practice is to declare the array as `const`. Otherwise, the API functions might create a copy of the array in anticipation of a possible copy-on-write operation.

See Also

`createCharArray`

C++ Cell Arrays

To create a cell array, use the `matlab::data::ArrayFactory createCellArray` function.

Create a `CellArray` that is equivalent to a MATLAB cell array defined with this MATLAB statement. Note that MATLAB assigns the cells in column-major order.

```
C = {'Character Array',...
    [true true false true];...
    [2.2 3.3 -4.2 6.0],...
    int32(-3374)};
```

Create an `ArrayFactory`:

```
matlab::data::ArrayFactory factory;
```

Call `createCellArray` and define each cell contained in the cell array:

```
matlab::data::CellArray C = factory.createCellArray({ 2,2 },
    factory.createCharArray("Character Array"),
    factory.createArray<double>({ 1, 4 }, { 2.2, 3.3, -4.2, 6.0}),
    factory.createArray<bool>({ 1, 4 }, { true, true, false, true }),
    factory.createScalar<int32_t>(-3374)
);
```

Modify the array by overwriting the value in the cell referred to in MATLAB as `C{1,1}`.

```
C[0][0] = factory.createCharArray("New Character Array");
```

Get a reference to the cell containing the double array and change the first element to `-2.2`.

```
TypedArrayRef<double> doubleArray = C[1][0];
doubleArray[0] = -2.2;
```

Display the new values in the cell containing the double array:

```
TypedArray<double> const newArray = C[1][0];
for (auto e : newArray) {
    std::cout << e << std::endl;
}
```

See Also

`matlab::data::ArrayFactory`

Related Examples

- “Access C++ Data Array Container Elements” on page 2-9

Access C++ Data Array Container Elements

The C++ MATLAB Data API `CellArray` and `StructArray` types are containers for other MATLAB Data Arrays. The elements in the containers are themselves arrays. There are two ways to access these elements:

- Get a reference to the elements of the container.
- Get a shared copy of the elements of the container.

Modify By Reference

To modify data in place, use a reference to the container element that you want to modify. For example, this code modifies the values of the first cell in the `CellArray` object. The first cell is a 1-by-3 logical array.

```
using namespace matlab::data;

ArrayFactory f;

auto cellArr = f.createCellArray({2,2},
    f.createArray<bool>({1,3},{true, true, false}),
    f.createCharArray("A char Array"),
    f.createScalar<int32_t>(-3374),
    f.createArray<double>({1,3},{2.2, 3.3, -4.2}));

// Get a reference to the first cell of the cell array.
TypedArrayRef<bool> ref = cellArr[0][0];

// Use the reference to modify the values in the cell.
for (auto& e : ref) {
    e = false;
}
```

After running this code, the first element of the cell array is a 1-by-3 logical array with each element set to `false`.

Copy Data from Container

You can access the data in a container using a shared copy. A shared copy enables you to get the data from the container or to modify the data in a copy that becomes nonshared when modified. Changing the data in a copy does not change the data in the container.

For example, this code creates a copy of the last cell in the `CellArray`, which is a 1-by-3 double array. The copy is modified by setting the first element in the double array to the numeric value 5.5. After this modification, the value in the `CellArray` is unchanged and the copy is no longer a shared value.

```
using namespace matlab::data;
ArrayFactory f;
auto cellArr = f.createCellArray({2,2},
    f.createArray<bool>({1,3},{true, true, false}),
    f.createCharArray("A cell Array"),
    f.createScalar<int32_t>(-3374),
    f.createArray<double>({1,3},{2.2, 3.3, -4.2}));
// Get a shared copy of the last element of the cell array.
TypedArray<double> cpy = cellArr[1][1];
cpy[0] = 5.5;
```

See Also

Related Examples

- “C++ Cell Arrays” on page 2-7

Operate on C++ Arrays Using Visitor Pattern

The C++ MATLAB Data API supports the use of visitor classes via the `matlab::data::apply_visitor` and `matlab::data::apply_visitor_ref` functions. These functions accept an array or array reference and a visitor class as inputs.

The `apply_visitor` and `apply_visitor_ref` functions dispatch to the operations defined by the visitor class based on input array type. The visitor class defines operations to perform on specific types of array.

Use the visitor pattern in cases such as these:

- There are many operations that you need to perform on an array and the way to perform them depends on the type of the array.
- The array returned by a function can be of different known types and you want to handle all cases.
- You are working with heterogeneous structures like cell arrays or structure arrays.

Dispatch on Array or Array Reference

The `apply_visitor` function dispatches to the visitor class operation based on the type of the input array. The syntax for calling `apply_visitor` accepts a `matlab::data::Array` and your visitor class instance:

```
auto apply_visitor(matlab::data::Array a, V visitor)
```

The `apply_visitor_ref` function dispatches to the visitor class operation based on the type of the array reference passed as an input. The syntax for calling `apply_visitor_ref` accepts a `matlab::data::ArrayRef` and your visitor class instance:

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

Overloading operator()

Implement your visitor class to overload the operator `operator()` for the array types you want to operate on. For example, suppose one operation that you want to implement is to return the text contained in a `matlab::data::CharArray` as a `std::string`. Implement the operation like this:

```
std::string operator()(matlab::data::CharArray arr){
    return arr.toAscii();
}
```

As another example, suppose that you want to negate the logical values in a `matlab::data::TypedArray`. In this case, use a reference to the array:

```
void operator()(TypedArrayRef<bool> boolArrRef) {
    std::cout << "Negate logical value: " << std::endl;
    for (auto &b : boolArrRef) {
        b = !b;
    }
}
```

You must use an element reference in the range-based for loop to change the value in the array.

Visitor Class to Display Contents of Cell Array

This example shows how to use a visitor class to define operations to perform on specific types of `matlab::data::Array`.

The `DisplayVisitor` class implements operations to display the contents of cell arrays for arrays of types `bool`, `double`, and `char`, and contained cell arrays. You can add new operations to support other cell array contents by adding more overloaded functions.

type [DisplayVisitor.cpp](#)

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

using namespace matlab::data;
void DisplayCell(const CellArray cellArray);

class DisplayVisitor {
public:
    template <typename U>
    void operator()(U arr) {}

    void operator()(const TypedArray<bool> boolArr) {
        std::cout << "Cell contains logical array: " << std::endl;
        for (auto b : boolArr) {
            printf_s("%d ", b);
        }
    }
}
```

```

        std::cout << "\n";
    }

    void operator()(const TypedArray<double> doubleArr) {
        std::cout << "Cell contains double array: " << std::endl;
        for (auto elem : doubleArr) {
            std::cout << elem << " ";
        }
        std::cout << "\n";
    }

    void operator()(const CharArray charArr) {
        std::cout << "Cell contains char array: " << std::endl;
        for (auto elem : charArr) {
            std::cout << char(elem);
        }
        std::cout << "\n";
    }

    void operator()(const CellArray containedCellArray) {
        DisplayCell(containedCellArray);
    }
};

void DisplayCell(const CellArray cellArray) {
    DisplayVisitor v;
    for (auto elem : cellArray) {
        apply_visitor(elem, v);
    }
}

```

To use the class, pass a cell array to the `DisplayCell` function.

type `callDisplayCell.cpp`

```

int main() {
    ArrayFactory factory;

    // Create cell array
    matlab::data::CellArray cellArray = factory.createCellArray({ 1,4 },
        factory.createCharArray("A char array"),
        factory.createArray<bool>({ 1,2 }, { false, true }),
        factory.createArray<double>({ 2,2 }, { 1.2, 2.2, 3.2, 4.2 }),
        factory.createCellArray({ 1,1 }, false));
}

```

```
    // Call function
    DisplayCell(cellArray);

    return 0;
}
```

Visitor Class to Modify Contents of Cell Array

In this example, the `CellModifyVisitor` class implements the operations to modify the contents of cell arrays of types `bool`, `double`, and `char`, and contained cell arrays. You can add new operations to support other cell array contents by adding more overloaded functions.

The `ModifyCell` function calls `apply_visitor_ref` in a loop for each element in the cell array. Because the objective is to modify the contents of the cell array, this example uses references to the cell array contents.

type `CellModifyVisitor.cpp`

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

using namespace matlab::data;
void ModifyCell(CellArray &cellArray);

class CellModifyVisitor {
public:
    template <typename U>
    void operator()(U arr) {}

    void operator()(TypedArrayRef<bool> boolArrRef) {
        std::cout << "Negate logical value: " << std::endl;
        for (auto &b : boolArrRef) {
            b = !b;
        }
    }

    void operator()(TypedArrayRef<double> doubleArrRef) {
        std::cout << "Add 1 to each value: " << std::endl;
        for (auto &elem : doubleArrRef) {
            elem = elem + 1;
        }
    }
}
```

```

        std::cout << "\n";
    }

    void operator()(CharArrayRef charArrRef) {
        std::cout << "Modify char array" << std::endl;
        ArrayFactory factory;
        charArrRef = factory.createCharArray("Modified char array");
    }

    void operator()(CellArrayRef containedCellArray) {
        CellModifyVisitor v;
        for (auto elem : containedCellArray) {
            apply_visitor_ref(elem, v);
        }
    }
};

void ModifyCell(CellArray &cellArray) {
    CellModifyVisitor v;
    for (auto elem : cellArray) {
        apply_visitor_ref(elem, v);
    }
}

```

To use the class, pass a cell array to the `ModifyCell` function.

type `callModifyCell.cpp`

```

int main() {
    ArrayFactory factory;

    // Create cell array
    matlab::data::CellArray cellArray = factory.createCellArray({ 1,4 },
        factory.createCharArray("A char array"),
        factory.createArray<bool>({ 1,2 }, { false, true }),
        factory.createArray<double>({ 2,2 }, { 1.2, 2.2, 3.2, 4.2 }),
        factory.createCellArray({ 1,1 }, false));

    // Call function
    ModifyCell(cellArray);

    return 0;
}

```

See Also

`matlab::data::apply_visitor` | `matlab::data::apply_visitor_ref`

Related Examples

- “C++ Cell Arrays” on page 2-7

MATLAB Data API Exceptions

In this section...

“matlab::data::CanOnlyUseOneStringIndexException” on page 2-17
“matlab::data::CantAssignArrayToThisArrayException” on page 2-18
“matlab::data::CantIndexIntoEmptyArrayException” on page 2-18
“matlab::data::DuplicateFieldNameInStructArrayException” on page 2-18
“matlab::data::FailedToLoadLibMatlabDataArrayException” on page 2-18
“matlab::data::FailedToResolveSymbolException” on page 2-18
“matlab::data::InvalidArrayIndexException” on page 2-18
“matlab::data::InvalidDimensionsInSparseArrayException” on page 2-18
“matlab::data::InvalidFieldNameException” on page 2-18
“matlab::data::MustSpecifyClassNameException” on page 2-19
“matlab::data::NonAsciiCharInRequestedAsciiOutputException” on page 2-19
“matlab::data::NonAsciiCharInInputDataException” on page 2-19
“matlab::data::InvalidArrayTypeException” on page 2-19
“matlab::data::NotEnoughIndicesProvidedException” on page 2-19
“matlab::data::StringIndexMustBeLastException” on page 2-19
“matlab::data::StringIndexNotValidException” on page 2-19
“matlab::data::SystemErrorException” on page 2-20
“matlab::data::TooManyIndicesProvidedException” on page 2-20
“matlab::data::TypeMismatchException” on page 2-20
“matlab::data::WrongNumberOfEnumsSuppliedException” on page 2-20
“matlab::data::InvalidMemoryLayout” on page 2-20
“matlab::data::InvalidDimensionsInRowMajorArrayException” on page 2-20
“matlab::data::NumberOfElementsExceedsMaximumException” on page 2-20

matlab::data::CanOnlyUseOneStringIndexException

The `CanOnlyUseOneStringIndexException` exception occurs if more than one string index is provided.

matlab::data::CantAssignArrayToThisArrayException

The `CantAssignArrayToThisArrayException` exception occurs if assigning an array to this array is not supported.

matlab::data::CantIndexIntoEmptyArrayException

The `CantIndexIntoEmptyArrayException` exception occurs when attempting any indexing operation on an empty array.

matlab::data::DuplicateFieldNameInStructArrayException

The `DuplicateFieldNameInStructArrayException` exception occurs if a duplicate field name is encountered in a struct definition.

matlab::data::FailedToLoadLibMatlabDataArrayException

The `FailedToLoadLibMatlabDataArrayException` exception occurs if necessary MATLAB Data Array libraries failed to load.

matlab::data::FailedToResolveSymbolException

The `FailedToResolveSymbolException` exception occurs if unable to resolve a required symbol in the MATLAB Data Array libraries.

matlab::data::InvalidArrayIndexException

The `InvalidArrayIndexException` exception occurs if the index provided is not valid for the array being indexed.

matlab::data::InvalidDimensionsInSparseArrayException

The `InvalidDimensionsInSparseArrayException` exception occurs if the caller attempts to create a sparse array with more than 2 dimensions.

matlab::data::InvalidFieldNameException

The `InvalidFieldNameException` exception occurs if field name is invalid for a struct.

matlab::data::MustSpecifyClassNameException

The `MustSpecifyClassNameException` exception occurs if class name is not specified.

matlab::data::NonAsciiCharInRequestedAsciiOutputException

The `NonAsciiCharInRequestedAsciiOutputException` exception occurs if user attempts to create a `CharArray` or a `StringArray` with a `std::string` and the `std::string` contains non-ascii characters.

matlab::data::NonAsciiCharInInputDataException

The `NonAsciiCharInInputDataException` exception occurs if user attempts to create a `CharArray` or a `StringArray` with a `std::string` and the `std::string` contains non-ascii characters.

matlab::data::InvalidArrayTypeException

The `InvalidArrayTypeException` exception occurs if the type of the rhs does not match the type of `TypedArray<T>`

matlab::data::NotEnoughIndicesProvidedException

The `NotEnoughIndicesProvidedException` exception occurs if not enough indices are provided.

matlab::data::StringIndexMustBeLastException

The `StringIndexMustBeLastException` exception occurs if a string index is not the last index.

matlab::data::StringIndexNotValidException

The `StringIndexNotValidException` exception occurs if a string index is not valid for this array.

matlab::data::SystemErrorException

The `SystemErrorException` exception occurs if a system error occurs.

matlab::data::TooManyIndicesProvidedException

The `TooManyIndicesProvidedException` exception occurs if too many indices are provided.

matlab::data::TypeMismatchException

The `TypeMismatchException` exception occurs if the element of the Array does not contain T's.

matlab::data::WrongNumberOfEnumsSuppliedException

The `WrongNumberOfEnumsSuppliedException` exception occurs if the wrong number of enums is provided.

matlab::data::InvalidMemoryLayout

The `InvalidMemoryLayout` exception occurs if you try to add a row-major array to a column-major buffer or vice versa.

matlab::data::InvalidDimensionsInRowMajorArrayException

The `InvalidDimensionsInRowMajorArrayException` exception occurs if the row-major array is not 2D.

matlab::data::NumberOfElementsExceedsMaximumException

The `NumberOfElementsExceedsMaximumException` exception occurs if the number of elements is greater than `size_t`.

See Also

MATLAB Data API Types

In this section...

“`matlab::data::ArrayDimensions`” on page 2-21
“`matlab::data::Enumeration`” on page 2-21
“`matlab::data::MATLABString`” on page 2-21
“`matlab::data::ObjectArray`” on page 2-21
“`matlab::data::String`” on page 2-22
“`matlab::data::Struct`” on page 2-22
“`buffer_ptr_t` and `buffer_deleter_t`” on page 2-22
“`iterator`” on page 2-22
“`const_iterator`” on page 2-22
“`reference`” on page 2-22
“`const_reference`” on page 2-22
“Reference Types” on page 2-22

`matlab::data::ArrayDimensions`

`ArrayDimensions` is defined as `std::vector<size_t>` in the `ArrayDimensions.hpp` header file.

`matlab::data::Enumeration`

`Enumeration` is defined in the `Enumeration.hpp` header file.

`matlab::data::MATLABString`

`MATLABString` is defined as `optional<String>` in the `String.hpp` header file.

`matlab::data::ObjectArray`

`ObjectArray` is defined as `TypedArray<Object>` in the `ObjectArray.hpp` header file.

matlab::data::String

String is defined as `std::basic_string<uchar>` in the `String.hpp` header file.

matlab::data::Struct

Struct is defined in the `Struct.hpp` header file.

buffer_ptr_t and buffer_deleter_t

`buffer_ptr_t` is defined as `std::unique_ptr<T[], buffer_deleter_t>`, where `buffer_deleter_t` is defined as `void (*)(void*)`.

iterator

`iterator` is defined as `TypedIterator<T>` in the `TypedArray.hpp` header file.

const_iterator

`const_iterator` is defined as `TypedIterator<typename std::add_const<T>::type>` in the `TypedArray.hpp` header file.

reference

`reference` is defined in the `TypedArray.hpp` header file as `typename iterator::reference`, where `iterator::reference` is `T&` for arithmetic types and `Reference<T>` for non-arithmetic types.

const_reference

`const_reference` is defined in the `TypedArray.hpp` header file as `typename const_iterator::reference`, where `const_iterator::reference` is `T&` for arithmetic types and `Reference<T>` for non-arithmetic types.

Reference Types

- `ArrayRef` is defined as `Reference<Array>` in the `TypedArrayRef.hpp` header file.

- `CellArrayRef` is defined as `Reference<TypedArray<Array>>` in the `TypedArrayRef.hpp` header file.
- `CharArrayRef` is defined as `TypedArrayRef<CHAR16_T>` in the `CharArray.hpp` header file.
- `EnumArrayRef` is defined as `TypedArrayRef<Enumeration>` in the `EnumArray.hpp` header file.
- `SparseArrayRef` is defined as `Reference<SparseArray<T>>` in the `SparseArrayRef.hpp` header file.
- `StructArrayRef` is defined as `Reference<TypedArray<Struct>>` in the `TypedArrayRef.hpp` header file.
- `TypedArrayRef` is defined as `Reference<TypedArray<T>>` in the `TypedArrayRef.hpp` header file.

Using Java Libraries from MATLAB

Getting Started with Java Libraries

From MATLAB, you can:

- Access Java class packages that support activities such as I/O and networking.
- Access third party Java classes.
- Construct Java objects in MATLAB workspace.
- Call Java object methods, using either Java or MATLAB syntax.
- Pass data between MATLAB variables and Java objects.

Platform Support for JVM Software

Every installation of MATLAB includes Java Virtual Machine (JVM™) software. To create and run programs that access Java objects, use the Java interpreter via MATLAB commands.

To find out which version of JVM software MATLAB uses on your platform, type the following at the MATLAB prompt:

```
version -java
```

For information about JVM support, see MATLAB Supported Language Interfaces.

Learn More About Java Programming Language

For a complete description of the Java language and for guidance in object-oriented software design and programming, consult outside resources. One resource is the help documentation on the www.oracle.com website.

See Also

Related Examples

- “Call Java Method” on page 3-4

External Websites

- MATLAB Supported Language Interfaces

Call Java Method

This example shows how to call a method of the `java.util.ArrayList` class. The example demonstrates what it means to have Java objects as references in MATLAB.

The `java.util.ArrayList` class is part of the Java standard libraries. Therefore, the class is already on the Java class path. If you call a method in a class that is not in a standard library, then update the Java class path so that MATLAB can find the method. For information, see “Java Class Path” on page 3-10.

Choose Class Constructor to Create Java Object

Create an `ArrayList` object by using one of the class constructors. Display the class methods and look for the `ArrayList` entries in the methods window.

```
methodsview('java.util.ArrayList')  
  
ArrayList    (java.util.Collection)  
ArrayList    ( )  
ArrayList    (int)
```

Choose the `ArrayList()` syntax, which constructs an empty list with an initial capacity of 10.

Shorten Class Name

Use the `import` function to refer to the `ArrayList` class without specifying the entire package name `java.util`.

```
import java.util.ArrayList
```

Create Array List

Create an empty `ArrayList` object.

```
A = ArrayList;
```

Pass MATLAB Data to add Method

Add items to the `ArrayList` object. Look in the methods window at the signatures for the `add` method.

```
void    add (int, java.lang.Object)
boolean add (java.lang.Object)
```

Choose the `boolean add (java.lang.Object)` syntax. The argument `java.lang.Object` is a Java type. To find the corresponding MATLAB type, look at the “Pass `java.lang.Object`” on page 3-61 table. If you pass a `double` argument, MATLAB converts it to a `java.lang.Double` type.

Add Elements to ArrayList

To call the `add` method, use MATLAB syntax.

```
add(A,5);
A
```

```
A =
```

```
[5.0]
```

Alternatively, use Java syntax.

```
A.add(10);
A
```

```
A =
```

```
[5.0, 10.0]
```

Java Objects Are References in MATLAB

To observe the behavior of copying Java objects, assign `A` to a new variable `B`.

```
B = A;
```

`B` is a reference to `A`. Any change to the object referenced by `B` also changes the object at `A`. Either MATLAB code or Java code can change the object. For example, add a value to `B`, and then display `A`.

```
add(B,15);  
A  
  
A =  
  
[5.0, 10.0, 15.0]
```

Use ArrayList Object in MATLAB

Suppose that you call a Java method that returns a Java object of type `ArrayList`. If you invoked the commands in the previous sections, variable `A` contains the following values:

```
class(A)  
  
ans =  
  
    'java.util.ArrayList'  
  
A  
  
A =  
  
[5.0, 10.0, 15.0]
```

To use `A` in MATLAB, convert the object to either a `java.lang.Object` type or to a primitive type. Then apply the MATLAB `cell` and `cell2mat` functions.

From the `ArrayList` methods window, find the `toArray` method that converts an `ArrayList` to `java.lang.Object[]`.

```
java.lang.Object[]    toArray    (java.lang.Object[])
```

Convert `A` to `java.lang.Object`.

```
res = toArray(A)  
  
res =  
  
    java.lang.Object[]:  
  
    [ 5]  
    [10]  
    [15]
```

Convert the output to a MATLAB type.

```
res = cell(res)'  
res =  
    1×3 cell array  
    [5]    [10]    [15]
```

To convert this value to a matrix, the elements must be the same type. In this example, the values convert to type double.

```
data = cell2mat(res)  
data =  
     5     10     15
```

See Also

`import` | `javaMethod`

Related Examples

- “Call Method in Your Own Java Class” on page 3-19

More About

- “Java Class Path” on page 3-10
- “Pass Data to Java Methods” on page 3-58
- “Handle Data Returned from Java Methods” on page 3-64
- “How MATLAB Allocates Memory”

External Websites

- <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>

Simplify Java Class Names Using import Function

MATLAB commands can refer to any Java class by its fully qualified name, which includes its package name. For example, the following are fully qualified names:

- `java.lang.String`
- `java.util.Enumeration`

A fully qualified name can be long, making commands and functions (such as constructors) cumbersome to edit and to read. To refer to classes by the class name alone (without a package name), first import the fully qualified name into MATLAB.

MATLAB adds all classes that you import to a list called the import list. To see what classes are on that list, type `import`. Your code can refer to any class on the list by class name alone.

When called from a function, `import` adds the specified classes to the import list in effect for that function. When invoked at the command prompt, `import` uses the base import list for your MATLAB platform.

For example, suppose that a function contains these statements:

```
import java.lang.String
import java.util.* java.awt.*
import java.util.Enumeration
```

The following statements refer to the `String`, `Frame`, and `Enumeration` classes without using the package names.

```
str = String('hello');    % Create java.lang.String object
frm = Frame;              % Create java.awt.Frame object
methods Enumeration      % List java.util.Enumeration methods
```

To remove the list of imported Java classes, type:

```
clear import
```

See Also

`import`

Related Examples

- “Call Java Method” on page 3-4

Java Class Path

To make Java classes available to MATLAB, place them on the Java class path. The class path is a series of file and folder specifications. When loading a Java class, MATLAB searches the files and folders in the order they occur on the class path. The search ends when MATLAB finds a file that contains the class definition.

Built-in Java class packages—classes in the Java standard libraries—are already on the class path. You do not need to modify the path to access these classes.

To access Java classes from MATLAB, add them to the class path. For information and examples, see “Static Path” on page 3-12.

- JAR File Classes
- Packages
- Individual (unpackaged) classes

MATLAB segments the Java class path into a static path and a dynamic path. MATLAB searches the static path before the dynamic path.

- Use the static path as the default path to load a Java class.
- Use the dynamic path when developing your own Java classes. You can modify and load the dynamic path any time during a MATLAB session.

Java Class Path Options	Action
Display class path	Call the <code>javaclasspath</code> function.
Add files to static path	Create an ASCII text file named <code>javaclasspath.txt</code> in your preferences folder. For information and examples, see “Static Path” on page 3-12.
Add or remove files on dynamic path	Call the <code>javaclasspath</code> , <code>javaaddpath</code> , or <code>javarmppath</code> functions. These functions clear all existing variables and global variables in the workspace. For more information, see “Dynamic Path” on page 3-18.

Java Class Path Options	Action
Augment search path for native method libraries.	Create an ASCII text file named <code>java.library.path.txt</code> in your preferences folder. For information, see “Locate Native Method Libraries” on page 3-16.

See Also

`java.class.path`

Related Examples

- “Call Java Method” on page 3-4
- “Call Method in Your Own Java Class” on page 3-19

More About

- “Static Path” on page 3-12
- “Dynamic Path” on page 3-18
- “Locate Native Method Libraries” on page 3-16

Static Path

The static path is loaded at the start of each MATLAB session from the MATLAB built-in Java path and the `javaclasspath.txt` file. The static path offers better Java class-loading performance than the dynamic Java path. However, if you modify the static path, you must restart MATLAB.

To add files to the static Java class path, create a `javaclasspath.txt` file. For instructions, see “Create `javaclasspath.txt` File” on page 3-12.

For convenience when developing your own Java classes, add entries to the dynamic Java class path. For information, see “Dynamic Path” on page 3-18.

For more information about how MATLAB uses the class path, see “Java Class Path” on page 3-10.

Create `javaclasspath.txt` File

Each line in the `javaclasspath.txt` file contains a reference to a Java class folder or JAR file. To create the file:

- 1 Create an ASCII text file named `javaclasspath.txt`.
- 2 Enter the name of a Java class folder or JAR file, one per line. The format of the name depends on how the class is defined.
 - For classes defined in Java packages, see “Add Packages” on page 3-13.
 - For classes defined in individual `.class` files, see “Add Individual (Unpackaged) Classes” on page 3-13.
 - For classes defined in Java Archive (JAR) files, see “Add JAR File Classes” on page 3-13.
- 3 Simplify folder specifications in cross-platform environments by using the `$matlabroot`, `$arch`, and `$jre_home` macros.
- 4 Save the file in your preferences folder. To view the location of the preferences folder, type:

```
prefdir
```

Alternatively, save the `javaclasspath.txt` file in your MATLAB startup folder. To identify the startup folder, type `pwd` at the command line immediately after starting

MATLAB. Classes specified in the `javaclasspath.txt` file in the startup folder appear on the path before classes specified in the file in the preferences folder. If a class appears in more than one folder or jar file, then Java uses the first one it finds.

5 Restart MATLAB.

MATLAB reads the static class path only at startup. If you edit `javaclasspath.txt` or change your `.class` files while MATLAB is running, then restart MATLAB to put those changes into effect.

If you do not want MATLAB to use the entries in the `javaclasspath.txt` file, then start MATLAB with the `-nouserjavapath` option.

For information about using the dynamic class path when writing your own Java classes, see “Dynamic Path” on page 3-18. For information about the startup folder, see “MATLAB Startup Folder”.

Add Individual (Unpackaged) Classes

To make individual classes—classes that are not part of a package—available in MATLAB, specify the full path to the folder containing the `.class` files. For example, for a compiled Java class in the file `c:\work\javaclasses\test.class`, add the following entry to the `javaclasspath.txt` file.

```
c:\work\javaclasses
```

Add Packages

To make a package available to MATLAB, specify the full path to the *parent folder of the highest level folder* of the package path. This folder is the first component in the package name. For example, if your Java class package `com.mw.tbx.ini` has its classes in folder `c:\work\com\mw\tbx\ini`, add the following entry to the `javaclasspath.txt` file.

```
c:\work
```

Add JAR File Classes

A JAR file contains multiple Java classes and packages in a compressed ZIP format. For information on the jar (Java Archive) tool and JAR files, consult your Java development documentation.

To make the contents of a JAR file available for use in MATLAB, specify the full path, *including the full file name*, for the JAR file. You also can put the JAR file on the MATLAB path.

Note The path requirement for JAR files is different from the requirement for `.class` files and packages, for which you do not specify file names.

For example, you have a JAR file named *mylibrary.jar* in the folder `C:\Documents\MATLAB\`, containing a method, *package.class.mymethod(params)*.

- Edit the `javaclasspath.txt` file.

```
cd(prefdir)
edit javaclasspath.txt
```

- Add the following text on a new line in the file.

```
C:\Documents\MATLAB\mylibrary.jar
```

- Save and close the file.
- Restart MATLAB.
- Call `mymethod`.

```
package.class.mymethod(params)
```

See Also

More About

- “Java Class Path” on page 3-10
- “Dynamic Path” on page 3-18
- “MATLAB Startup Folder”

External Websites

- Java™ Platform, Standard Edition 7 API Specification

Load Java Class Definitions

MATLAB loads a Java class automatically when your code first uses it, for example, when you call its constructor. To display a list of currently loaded Java classes, call the `inmem` function. This function returns a list of classes in the output argument `J`. For example:

```
[~,~,j] = inmem;  
j  
  
j =  
  
    2x1 cell array  
  
'java.util.Date'  
'com.mathworks.ide.desktop.MLDesktop'
```

MATLAB displays what is loaded on your system.

Note When you call the `which` function on methods defined by Java classes, the function only displays the classes currently loaded into the MATLAB workspace. In contrast, `which` displays MATLAB classes, whether or not they are loaded.

See Also

`inmem`

Locate Native Method Libraries

Java classes can dynamically load native methods using the Java method `java.lang.System.loadLibrary("LibFile")`. To load the library file *LibFile*, the folder containing it must be on the Java Library Path. The JVM software defines this path at startup.

You can augment the search path for native method libraries by creating an ASCII text file named `javalibrarypath.txt` in your preferences folder. Follow these guidelines when editing this file.

- Create the file.

```
cd(prefdir)
edit javalibrarypath.txt
```

- Specify each new folder on a line by itself.
- Specify only the folder names, not the names of the DLL files. The `System.loadLibrary` call reads the file names.
- In cross-platform environments, simplify the specification of folders by using the `$matlabroot`, `$arch`, and `$jre_home` macros.

You also can create a `javalibrarypath.txt` file in your MATLAB startup folder. To identify the startup folder, type `pwd` at the command line immediately after starting MATLAB. Libraries specified in the `javalibrarypath.txt` file in the startup folder override libraries specified in the file in the preferences folder.

To disable using the `javalibrarypath.txt` file, start MATLAB with the `-nouserjavapath` option.

See Also

`prefdir`

More About

- “MATLAB Startup Folder”

Load Class Using Java Class.forName Method

Instead of using the Java `Class.forName` method, call the MATLAB `javaObjectEDT` function. For example, replace this statement:

```
java.lang.Class.forName('xyz.myapp.MyClass')
```

with:

```
javaObjectEDT('xyz.myapp.MyClass')
```

See Also

`javaObjectEDT`

Dynamic Path

MATLAB segments the Java class path into a static path and a dynamic path. MATLAB searches the static path before the dynamic path.

MATLAB provides the dynamic path as a convenience for when you develop your own Java classes. You can change class definitions on the dynamic path without restarting MATLAB. Therefore, it is useful to put a user-defined Java class definition on the dynamic path while you develop and debug the class.

While the dynamic path offers greater flexibility in changing the path, Java classes on the dynamic path might load more slowly than classes on the static path. Also, classes on the dynamic path might not behave identically to classes on the static path. If your class does not behave as expected, then use the static path.

After developing a Java class, put the class on the static path. For more information, see “Static Path” on page 3-12

To add a class to the dynamic path, use the `javaclasspath` and `javaaddpath` functions. To remove an entry, use the `javarmpath` function. These functions clear all existing variables and global variables in the workspace.

Note Do not put Java classes on the static path if they have dependencies on classes on the dynamic path.

See Also

`javaaddpath` | `javaclasspath` | `javarmpath`

More About

- “Static Path” on page 3-12

Call Method in Your Own Java Class

To define new Java classes and subclasses of existing classes, use a Java Development Kit external to MATLAB. For information on supported versions of JDK™ software, see the MATLAB Supported Language Interfaces website.

After you create class definitions in `.java` files, use your Java compiler to produce `.class` files. The next step is to make the class definitions in those `.class` files available for you to use in MATLAB.

This example shows how to call a method in your own Java class. The example uses a class file named `myclass.class` in the folder `C:\Documents\MATLAB\` containing a method `package.myclass.mymethod(params)`.

Put the class file on the dynamic Java class path, making the class available in the current MATLAB session only. MATLAB provides the dynamic path as a convenience for when you develop your own Java classes.

- Add the class to the dynamic Java class path. To access the class, you must modify the Java path every time you start MATLAB.

```
javaaddpath('C:\Documents\MATLAB\')
```

- Call the method. Substitute your own class file name and path, and call the method with the appropriate parameter list.

```
package.myclass.mymethod(params)
```

- Make the class always available in MATLAB. Add it to the static class path by editing the `javaclasspath.txt` file in your `prefdir` folder.

See Also

`javaaddpath`

More About

- “Dynamic Path” on page 3-18
- “Static Path” on page 3-12

External Websites

- MATLAB Supported Language Interfaces

How MATLAB Represents Java Arrays

The term Java array refers to a container object that holds a fixed number of values of a single type. The type of an array is written as `type []`. An array of arrays—also known as a multidimensional array—uses two or more sets of brackets, such as `String [][]`.

The term *dimension* refers to the number of subscripts required to address the elements of an array. Dimension is not a measure of length, width, and height. For example, a 5-by-1 array is one-dimensional, because you use one subscript to access an individual element. To work with a two-dimensional array, create an array of arrays. To add further dimensions, add more levels to the array, making it an array of arrays of arrays, and so on.

MATLAB treats multilevel Java arrays like matrices and multidimensional arrays. Use the same MATLAB syntax to access elements of a Java array.

Array Indexing

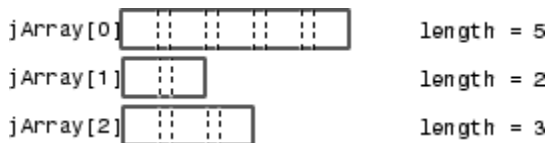
Java array indices are zero-based while MATLAB array indices are one-based. In Java programming, you access the elements of array `y` of length `N` using `y[0]` through `y[N-1]`. When working with this array in MATLAB, you access these elements using `y(1)` through `y(N)`.

For an example, see “Access Elements of Java Array” on page 3-26.

Shape of Java Arrays

A two-dimensional MATLAB array is a rectangle, as each row is of equal length and each column of equal height. A Java array is an array of arrays and does not necessarily hold to this rectangular form. Each individual lower-level array might have a different length.

The following picture shows an array of three underlying arrays of different lengths. The term *jagged* (or *ragged*) is commonly used to describe this arrangement of array elements as the array ends do not match up evenly. When a Java method returns a jagged array of primitive Java types, MATLAB stores it in a cell array.



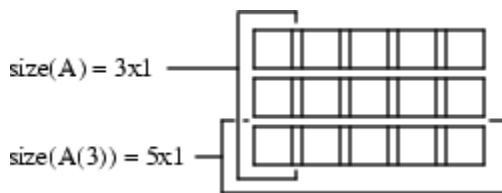
The MATLAB `string` function pads a jagged Java string array, making it a rectangular MATLAB array.

Interpret Size of Java Arrays

The MATLAB `size` function returns the length of the Java array. The number of columns is always 1.

The potentially ragged shape of a Java array makes it impossible to size the array in the same way as for a MATLAB array. In a Java array, no single value represents the size of the lower-level arrays.

For example, consider the following Java array.



`size(A)` returns the dimensions of the highest array level of `A`. The highest level of the array has a size of 3-by-1.

```
size(A)
```

```
ans =  
     3     1
```

To find the size of a lower-level array, for example the five-element array in row 3, refer to the row explicitly.

```
size(A(3))
```

```
ans =  
     5     1
```

You can specify a dimension in the `size` command using the following syntax. However, this command only sizes the first dimension, `dim=1`, the only nonunity dimension.

```
m = size(X,dim)  
size(A,1)
```

```
ans =  
    3
```

Interpret Number of Dimensions of Java Arrays

The MATLAB `ndims` function always returns a value of 2 for the number of dimensions in a Java array. This value is the number of dimensions in the top-level array.

Display Java Vector

MATLAB displays a Java vector as a column but processes it as if it were a row vector. For examples, see “Concatenate Java Arrays” on page 3-35.

See Also

`ndims` | `size`

Create Array of Java Objects

The MATLAB `javaArray` function lets you create a Java array that MATLAB handles as a single multidimensional array. You specify the number and size of the array dimensions along with the class of objects you intend to store in it. Using the one-dimensional Java array as its primary building block, MATLAB then builds a Java array that satisfies the dimensions requested in the `javaArray` command.

To create a Java object array, use the MATLAB `javaArray` function. For example, the following command creates a Java array of four lower-level arrays, each containing five objects of the `java.lang.Double` class.

```
dblArray = javaArray('java.lang.Double',4,5);
```

The `javaArray` function does not initialize values in the array. This code copies the first four rows of MATLAB array `A`, containing randomly generated data, into `dblArray`.

```
A = rand(5);  
for m = 1:4  
    for n = 1:5  
        dblArray(m,n) = java.lang.Double(A(m,n));  
    end  
end
```

```
dblArray
```

```
dblArray =
```

```
java.lang.Double[][]:  
  
    [0.7577]    [0.7060]    [0.8235]    [0.4387]    [0.4898]  
    [0.7431]    [0.0318]    [0.6948]    [0.3816]    [0.4456]  
    [0.3922]    [0.2769]    [0.3171]    [0.7655]    [0.6463]  
    [0.6555]    [0.0462]    [0.9502]    [0.7952]    [0.7094]
```

You must convert each element of `A` to the `java.lang.Double` type. For more information, see “Pass Java Objects” on page 3-61.

Create Array of Primitive Java Types

To pass an array of a primitive Java type to a Java method, you must pass in an array of the equivalent MATLAB type. For the type mapping details, see “MATLAB Type to Java Type Mapping” on page 3-58.

For example, create a `java.awt.Polygon` by looking at the constructors in the following methods window.

```
methodsview('java.awt.Polygon')
```

This constructor uses an array of Java `int`.

```
Polygon    (int[],int[],int)
```

MATLAB converts a MATLAB `double` to a Java scalar or array `int`. Create two MATLAB arrays, identifying four points of a polygon.

```
x = [10 40 95 125 10];  
y = [50 15 0 60 50];  
polygon = java.awt.Polygon(x,y,length(x));
```

To call the `Polygon` object method `contains`, look at its signature in the method window.

```
boolean    contains    (double,double)
```

MATLAB converts a MATLAB `double` to a Java `double`. This statement checks if the point (50,40) is within the polygon.

```
contains(polygon,50,40)
```

```
ans =
```

```
logical
```

```
1
```

See Also

`javaArray`

More About

- “Pass Java Objects” on page 3-61
- “MATLAB Type to Java Type Mapping” on page 3-58

Access Elements of Java Array

MATLAB Array Indexing

To access elements of a Java object array, use the MATLAB array indexing syntax, `A(row, column)`. In a Java program, the syntax is `A[row-1][column-1]`.

Single Subscript Indexing

When you refer to the elements of a MATLAB matrix with a single subscript, MATLAB returns a single element of the matrix. In contrast, single subscript (linear) indexing into a multidimensional Java array returns a subarray.

For example, create a MATLAB array.

```
for m = 1:4
    for n = 1:5
        matlabArr(m,n) = (m*10)+n;
    end
end
matlabArr
```

matlabArr =

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45

Copy the contents into a Java array.

```
javaArr = javaArray('java.lang.Integer',4,5);
for m = 1:4
    for n = 1:5
        javaArr(m,n) = java.lang.Integer(matlabArr(m,n));
    end
end
javaArr
```

javaArr =

```
java.lang.Integer[][]:
```

```

[11] [12] [13] [14] [15]
[21] [22] [23] [24] [25]
[31] [32] [33] [34] [35]
[41] [42] [43] [44] [45]

```

Index value 3 returns a single element in the MATLAB array.

```
matlabArr(3)
```

```
ans = 31
```

Index value 3 returns the entire third row in a Java array.

```
javaArr(3)
```

```
ans =
```

```
java.lang.Integer[]:
```

```

[31]
[32]
[33]
[34]
[35]

```

Linear indexing into a Java array enables you to specify an entire array from a larger array structure. Then you can manipulate it as an object.

Colon Operator Indexing

To specify a range of elements in an array, use the colon operator (:). For example, create a 4-by-5 Java array.

```

dblArray = javaArray('java.lang.Double',4,5);
for m = 1:4
    for n = 1:5
        dblArray(m,n) = java.lang.Double((m*10)+n);
    end
end
dblArray

dblArray =

```

```
java.lang.Double[][]:  
  
[11] [12] [13] [14] [15]  
[21] [22] [23] [24] [25]  
[31] [32] [33] [34] [35]  
[41] [42] [43] [44] [45]
```

Create a subarray `row2Array` from the elements in columns 2 through 4.

```
row2Array = dblArray(2,2:4)
```

```
row2Array =
```

```
java.lang.Double[]:  
  
[22]  
[23]  
[24]
```

You also can use the colon with linear indexing to refer to all elements in the entire matrix. However, Java and MATLAB arrays are stored differently in memory meaning that the order of the elements in a linear array is different. Java array elements are stored in a row-by-column format, an order that matches the rows of the matrix. MATLAB array elements are stored column-wise, an order that matches the columns. For example, convert the 4-by-5 array `dblArray` into a 20-by-1 linear array.

```
linearArray = dblArray(:)
```

```
linearArray =
```

```
java.lang.Double[]:  
  
[11]  
[12]  
[13]  
[14]  
[15]  
[21]  
[22]  
[23]  
[24]  
[25]  
[31]  
[32]  
[33]
```

```
[34]
[35]
[41]
[42]
[43]
[44]
[45]
```

Using END in a Subscript

To reference the top-level array in a multilevel Java array, use the `end` keyword as the first subscript. For example, display data from the third to the last rows of Java array `dblArray`.

```
last2rows = dblArray(3:end, :)
```

```
last2rows =
```

```
java.lang.Double[][]:
```

```
  [31]  [32]  [33]  [34]  [35]
  [41]  [42]  [43]  [44]  [45]
```

Do not use `end` on lower-level arrays. Because of the potentially ragged nature of the arrays, MATLAB cannot determine the end value. For more information, see “Shape of Java Arrays” on page 3-21.

Converting Object Array Elements to MATLAB Types

When you access an element of a `java.lang.Object` array, MATLAB converts the element to a MATLAB type, based on the table in “`java.lang.Object` Return Types” on page 3-65. MATLAB does not convert elements of any other type of Java array.

For example, if a `java.lang.Object` array contains a `java.lang.Double` element, then MATLAB converts the element to a MATLAB `double`. However, MATLAB does not convert a `java.lang.Double` element in a `java.lang.Double` array. MATLAB returns it as `java.lang.Double`.

See Also

More About

- “Shape of Java Arrays” on page 3-21
- “java.lang.Object Return Types” on page 3-65
- “Array Indexing”

Assign Values to Java Array

To assign values to objects in a Java object array, use the MATLAB command syntax. For example, the following statement assigns a value to Java array `A` of type `java.lang.Double`.

```
A(row,column) = java.lang.Double(value)
```

In a Java program, you would assign the value to `A[row-1][column-1]`. For more information on the differences between Java and MATLAB arrays, see “How MATLAB Represents Java Arrays” on page 3-21.

To run the examples in this topic, create a 4-by-5 array `dblArray`. The values displayed for `dblArray` depend on the order in which you run the examples.

```
dblArray = javaArray('java.lang.Double',4,5);
for m = 1:4
    for n = 1:5
        dblArray(m,n) = java.lang.Double((m*10)+n);
    end
end
dblArray
```

```
dblArray =
```

```
java.lang.Double[][]:
```

```
[11] [12] [13] [14] [15]
[21] [22] [23] [24] [25]
[31] [32] [33] [34] [35]
[41] [42] [43] [44] [45]
```

Single Subscript Indexing Assignment

You can use single-subscript indexing to assign values to an array. For example, create a 5-by-1 Java array and assign it to a row of `dblArray`.

```
onedimArray = javaArray('java.lang.Double',5);
for k = 1:5
    onedimArray(k) = java.lang.Double(100*k);
end
```

Replace row 3 with the values of `onedimArray`.

```
dblArray(3) = onedimArray
```

```
dblArray =
```

```
java.lang.Double[][]:
```

```
[ 11]    [ 12]    [ 13]    [ 14]    [ 15]
[ 21]    [ 22]    [ 23]    [ 24]    [ 25]
[100]    [200]    [300]    [400]    [500]
[ 41]    [ 42]    [ 43]    [ 44]    [ 45]
```

Linear Array Assignment

To assign a value to every element of a multidimensional Java array, use the MATLAB colon operator (:). For example, initialize the contents of `dblArray` to zero.

```
dblArray(:) = java.lang.Double(0)
```

```
dblArray =
```

```
java.lang.Double[][]:
```

```
[0]    [0]    [0]    [0]    [0]
[0]    [0]    [0]    [0]    [0]
[0]    [0]    [0]    [0]    [0]
[0]    [0]    [0]    [0]    [0]
```

Use the colon operator as you would when working with MATLAB arrays. For example, assign one value to each row in `dblArray`.

```
dblArray(1,:) = java.lang.Double(125);
```

```
dblArray(2,:) = java.lang.Double(250);
```

```
dblArray(3,:) = java.lang.Double(375);
```

```
dblArray(4,:) = java.lang.Double(500)
```

```
dblArray =
```

```
java.lang.Double[][]:
```

```
[125]    [125]    [125]    [125]    [125]
[250]    [250]    [250]    [250]    [250]
[375]    [375]    [375]    [375]    [375]
[500]    [500]    [500]    [500]    [500]
```


Empty Matrix Assignment

You can assign the empty matrix ([]) to a Java array element. MATLAB stores the null value, rather than a 0-by-0 array.

```
dblArray(2,2) = []
```

```
dblArray =
```

```
java.lang.Double[][]:
```

```

[125]    [125]    [125]    [125]    [125]
[250]     []    [250]    [250]    [250]
[375]    [375]    [375]    [375]    [375]
[500]    [500]    [500]    [500]    [500]
```

Subscripted Deletion

If you assign an empty matrix to an entire row or column of a MATLAB array, then MATLAB removes that row or column from the array. When you assign the empty matrix to a Java array, the array maintains its dimensions.

For example, create a MATLAB array.

```

for m = 1:4
    for n = 1:5
        matlabArr(m,n) = (m*10) + n;
    end
end
matlabArr
```

```

matlabArr =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45
```

Assign the empty matrix to the fourth column. This statement changes its dimensions from 4-by-5 to 4-by-4.

```
matlabArr(:,4) = []
```

```

matlabArr =
    11    12    13    15
```

```
21  22  23  25
31  32  33  35
41  42  43  45
```

When you assign the empty matrix to the Java array `dblArray`, the array maintains its 4-by-5 dimensions.

```
dblArray(:,4) = []
```

```
dblArray =
```

```
java.lang.Double[][]:
```

```
[125] [125] [125] [] [125]
[250] [] [250] [] [250]
[375] [375] [375] [] [375]
[500] [500] [500] [] [500]
```

Concatenate Java Arrays

To concatenate arrays of Java objects, use the MATLAB `cat` function or the square bracket (`[]`) operators.

You can concatenate Java objects only along the first (vertical) or second (horizontal) axis. For more information, see “How MATLAB Represents Java Arrays” on page 3-21.

Two-Dimensional Horizontal Concatenation

This example horizontally concatenates two Java arrays. Create 2-by-3 arrays `d1` and `d2`.

```
d1 = javaArray('java.lang.Double',2,3);
for m = 1:2
    for n = 1:3
        d1(m,n) = java.lang.Double(n*2 + m-1);
    end
end
d1

d1 =

    java.lang.Double[][]:

        [2]    [4]    [6]
        [3]    [5]    [7]

d2 = javaArray('java.lang.Double',2,2);
for m = 1:2
    for n = 1:3
        d2(m,n) = java.lang.Double((n+3)*2 + m-1);
    end
end
d2

d2 =

    java.lang.Double[][]:

        [8]    [10]   [12]
        [9]    [11]   [13]
```

Concatenate the two arrays along the second (horizontal) dimension.

```
d3 = cat(2,d1,d2)
d3 =
    java.lang.Double[][]:
        [2]    [4]    [6]    [8]    [10]    [12]
        [3]    [5]    [7]    [9]    [11]    [13]
```

Vector Concatenation

This example show the difference between row and column concatenation for vectors. Create two vectors J1 and J2.

```
import java.lang.Integer
J1 = [];
for ii = 1:3
    J1 = [J1;Integer(ii)];
end
J1
```

```
J1 =
    java.lang.Integer[]:
        [1]
        [2]
        [3]
```

```
J2 = [];
for ii = 4:6
    J2 = [J2;Integer(ii)];
end
J2
```

```
J2 =
    java.lang.Integer[]:
        [4]
        [5]
        [6]
```

Concatenate by column. Horizontally concatenating two Java vectors creates a longer vector, which prints as a column.

```
Jh = [J1,J2]
```

```
Jh =
```

```
java.lang.Integer[]:
```

```
[1]
[2]
[3]
[4]
[5]
[6]
```

Concatenate by row. Vertically concatenating two Java vectors creates a 2-D Java array.

```
Jv = [J1;J2]
```

```
Jv =
```

```
java.lang.Integer[][]:
```

```
[1] [2] [3]
[4] [5] [6]
```

Note Unlike MATLAB, a 3x1 Java array is not the same as a Java vector of length 3. Create a 3x1 array.

```
import java.lang.Integer
arr1 = javaArray('java.lang.Integer',3,1)
```

```
arr1 =
```

```
java.lang.Integer[][]:
```

```
[]
[]
[]
```

Create a vector of length 3.

```
arr2 = javaArray('java.lang.Integer',3)
```

```
arr2 =
```

```
java.lang.Integer[]:
```

[]
[]
[]

See Also

More About

- “Construct and Concatenate Java Objects” on page 3-43
- “Creating, Concatenating, and Expanding Matrices”
- “How MATLAB Represents Java Arrays” on page 3-21

Create Java Array References

Java arrays in MATLAB are *references*. Assigning an array variable to another variable results in a second reference to the array, not a copy of the array. For example, create and initialize `origArray`.

```
origArray = javaArray('java.lang.Double',3,4);
for m = 1:3
    for n = 1:4
        origArray(m,n) = java.lang.Double((m*10)+n);
    end
end
origArray
```

```
origArray =
```

```
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

Create a second reference to the array `newArrayRef`.

```
newArrayRef = origArray;
```

Change the array referred by `newArrayRef`. The changes also show up in `origArray`.

```
newArrayRef(3,:) = java.lang.Double(0);
origArray
```

```
origArray =
```

```
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [ 0]    [ 0]    [ 0]    [ 0]
```

See Also

Related Examples

- “Create Copy of Java Array” on page 3-41

Create Copy of Java Array

To make a copy of an existing Java array, use subscript indexing. For example, create and initialize `origArray`.

```
origArray = javaArray('java.lang.Double',3,4);
for m = 1:3
    for n = 1:4
        origArray(m,n) = java.lang.Double((m*10)+n);
    end
end
origArray
```

```
origArray =
```

```
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

Copy the entire contents to `newArray`.

```
newArray = origArray(:, :)
```

```
newArray =
```

```
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

Change elements of `newArray`.

```
newArray(3, :) = java.lang.Double(0)
```

```
newArray =
```

```
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [ 0]    [ 0]    [ 0]    [ 0]
```

The values in `origArray` do not change.

```
origArray
```

```
origArray =
```

```
java.lang.Double[][]:
```

```
    [11]    [12]    [13]    [14]  
    [21]    [22]    [23]    [24]  
    [31]    [32]    [33]    [34]
```

See Also

Related Examples

- “Create Java Array References” on page 3-39

Construct and Concatenate Java Objects

Create Java Object

Many Java method signatures contain Java object arguments. To create a Java object, call one of the constructors of the class. For an example, see “Call Java Method” on page 3-4.

Concatenate Objects of Same Class

To concatenate Java objects, use either the `cat` function or the `[]` operators.

Concatenating objects of the same Java class results in an array of objects of that class.

```
value1 = java.lang.Integer(88);  
value2 = java.lang.Integer(45);  
cat(1,value1,value2)
```

```
ans =
```

```
java.lang.Integer[]:  
  
[88]  
[45]
```

Concatenate Objects of Unlike Classes

If you concatenate objects of unlike classes, MATLAB finds one class from which all the input objects inherit. MATLAB selects the lowest common parent in the Java class hierarchy as the output class. For example, concatenating objects of `java.lang.Byte`, `java.lang.Integer`, and `java.lang.Double` creates an object of the common parent to the three input classes, `java.lang.Number`.

```
byte = java.lang.Byte(127);  
integer = java.lang.Integer(52);  
double = java.lang.Double(7.8);  
[byte integer double]
```

```
ans =
```

```
java.lang.Number[]:
```

```
[ 127]
[ 52]
[7.8000]
```

If there is no common, lower-level parent, then the resultant class is `java.lang.Object`.

```
byte = java.lang.Byte(127);
point = java.awt.Point(24,127);
[byte point]
```

```
ans =
```

```
java.lang.Object[]:
[ 127]
[1x1 java.awt.Point]
```

See Also

Related Examples

- “Call Java Method” on page 3-4

Save and Load Java Objects to MAT-Files

To save or serialize a Java object to a MAT-file, call the `save` function. To load the object from a MAT-file back into MATLAB, call the `load` function.

When you save or load a Java object, the object and its class must meet all this criteria:

- The class implements the Java API `Serializable` interface, either directly or by inheriting it from a parent class. Any embedded or otherwise referenced objects also must implement `Serializable`.
- Do *not* change the class definition between saving and loading the object. Changes to the data fields or methods of a class prevent the loading of an object that was constructed with another class definition.
- Values in transient data fields are not saved with the object. Either the class does not have any transient data fields, or the values are not significant.

If you define your own Java classes, or subclasses of existing classes, follow the same criteria to enable saving and loading objects of the class in MATLAB. For details on defining classes to support serialization, consult your Java development documentation.

See Also

`load` | `save`

Data Fields of Java Objects

Access Public and Private Data

Java classes can contain member variables called fields which might have public or private access.

To access public data fields, which your code can read or modify directly, use the syntax:

```
object.field
```

To read from and, where allowed, to modify private data fields, use the accessor methods defined by the Java class. These methods are sometimes referred to as *get* and *set* methods.

For example, the `java.awt.Frame` class has both private and public data fields. The read accessor method `getSize` returns a `java.awt.Dimension` object.

```
frame = java.awt.Frame;  
frameDim = getSize(frame)  
  
frameDim =  
  
java.awt.Dimension[width=0,height=0]
```

The `Dimension` class has public data fields `height` and `width`. Display the value of `height`.

```
height = frameDim.height  
  
height = 0
```

Set the value of `width`.

```
frameDim.width = 42  
  
frameDim =  
  
java.awt.Dimension[width=42,height=0]
```

Display Public Data Fields of Java Object

To list the public fields of a Java object, call the `fieldnames` function. For example, create an `Integer` object and display the field names.

```
value = java.lang.Integer(0);
fieldnames(value)
```

```
ans =
  'MIN_VALUE'
  'MAX_VALUE'
  'TYPE'
  'SIZE'
```

To display more information about the data fields, type:

```
fieldnames(value, '-full')
```

```
ans =
  'static final int MIN_VALUE'
  'static final int MAX_VALUE'
  'static final java.lang.Class TYPE'
  'static final int SIZE'
```

Access Static Field Data

A static data field is a field that applies to an entire class of objects. To access static fields, use the class name. For example, display the `TYPE` field of the `Integer` class.

```
thisType = java.lang.Integer.TYPE
```

```
thisType =
```

```
int
```

Alternatively, create an instance of the class.

```
value = java.lang.Integer(0);
thatType = value.TYPE
```

```
thatType =
```

```
int
```

MATLAB does not allow assignment to static fields using the class name. To assign a value, use the static `set` method of the class or create an instance of the class. For example, assign `value` to the following `staticFieldName` field by creating an instance of `java.className`.

```
objectName = java.className;  
objectName.staticFieldName = value;
```

See Also

`fieldnames`

Determine Class of Java Object

To find the class of a Java object, use the `class` function. For example:

```
value = java.lang.Integer(0);  
myClass = class(value)
```

```
myClass = java.lang.Integer
```

The `class` function does not tell you whether the class is a Java class. For this information, call the `isjava` function. For example, the class `value` is a Java class:

```
isjava(value)
```

```
ans =  
    1
```

To find out if an object is an instance of a specific class, call the `isa` function. The class can be a MATLAB built-in, a user-defined class, or a Java class. For example:

```
isa(value, 'java.lang.Integer')
```

```
ans =  
    1
```

See Also

`class` | `isa` | `isjava`

Method Information

Display Method Names

The `methods` function returns information on methods of MATLAB and Java classes.

To return the names of all the methods (including inherited methods) of the class, use `methods` without the `'-full'` qualifier. Names of overloaded methods are listed only once.

Display Method Signatures

With the `'-full'` qualifier, `methods` returns a listing of the method names (including inherited methods) along with attributes, argument lists, and inheritance information on each. Each overloaded method is listed separately.

For example, display a full description of all methods of the `java.awt.Dimension` object.

```
methods java.awt.Dimension -full
```

```
Methods for class java.awt.Dimension:
Dimension()
Dimension(java.awt.Dimension)
Dimension(int,int)
java.lang.Class getClass() % Inherited from java.lang.Object
int hashCode() % Inherited from java.lang.Object
boolean equals(java.lang.Object)
java.lang.String toString()
void notify() % Inherited from java.lang.Object
void notifyAll() % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
    % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
    % Inherited from java.lang.Object
void wait() throws java.lang.InterruptedException
    % Inherited from java.lang.Object
java.awt.Dimension getSize()
void setSize(java.awt.Dimension)
void setSize(int,int)
```

Display Syntax in Figure Window

To see methods implemented by a particular Java (or MATLAB) class, use the `methodsview` function. Specify the class name (along with its package name, for Java classes) in the command line. If you have imported the package that defines this class, then the class name alone suffices.

This command lists information on all methods in the `java.awt.MenuItem` class:

```
methodsview java.awt.MenuItem
```

A new window appears, listing one row of information for each method in the class. Each row in the window displays up to six fields of information describing the method. This table lists the fields displayed in the `methodsview` window along with a description and examples of each field type.

Fields Displayed in `methodsview` Window

Field Name	Description	Examples
Qualifiers	Method type qualifiers	<code>abstract</code> , <code>synchronized</code>
Return Type	Type returned by the method	<code>void</code> , <code>java.lang.String</code>
Name	Method name	<code>addActionListener</code> , <code>dispatchEvent</code>
Arguments	Types of arguments passed to method	<code>boolean</code> , <code>java.lang.Object</code>
Other	Other relevant information	throws <code>java.io.IOException</code>
Inherited From	Parent of the specified class	<code>java.awt.MenuComponent</code>

See Also

`methods` | `methodsview`

Determine What Classes Define a Method

To display the fully qualified name of a method implemented by a *loaded* Java class, call the `which` function. To find all classes that define the specified method, use `which` with the `-all` qualifier.

For example, to find the package and class name for the `concat` method, type:

```
which concat
```

If the `java.lang.String` class is loaded, MATLAB displays:

```
concat is a Java method % java.lang.String method
```

If the `String` class has not been loaded, MATLAB displays:

```
concat not found.
```

Suppose that you loaded the Java `String` and `java.awt.Frame` classes. Both of these classes have an `equals` method. Type:

```
which -all equals
```

The MATLAB display includes entries like these:

```
equals is a Java method      % java.lang.String method
equals is a Java method      % java.awt.Frame.equals
equals is a Java method      % com.mathworks.jmi.MatlabPath method
```

The `which` function operates differently on Java classes than it does on MATLAB classes:

- `which` always displays MATLAB classes, whether they are loaded
- `which` only displays Java classes that are loaded

You can find out which Java classes are currently loaded by using the command `[m,x,j]=inmem`.

See Also

`inmem` | `which`

More About

- “Load Java Class Definitions” on page 3-15
- “Java Class Path” on page 3-10

Java Methods That Affect MATLAB Commands

MATLAB commands that operate on Java objects and arrays use the methods that are implemented within, or inherited by, the class. There are some MATLAB commands that you can alter in behavior by changing the Java methods that they use.

Changing the Effect of `disp` and `display`

You call the `disp` function when you:

- Display the value of a variable or an expression in MATLAB.
- Terminate a command line without a semicolon.
- Display a Java object in MATLAB.

When calling `disp` on a Java object, MATLAB formats the output using the object `toString` method. If the class does not implement this method, then MATLAB uses an inherited `toString` method. If no intermediate ancestor classes define this method, MATLAB uses the `toString` method defined by the `java.lang.Object` class.

To change the way MATLAB displays an object, implement your own `toString` method in your class definition.

Changing the Effect of `isequal`

The MATLAB `isequal` function compares two or more arrays for equality in type, size, and contents. Also, you can use this function to test Java objects for equality.

When you compare two Java objects using `isequal`, MATLAB performs the comparison using the Java method, `equals`. MATLAB first determines the class of the objects specified in the command, and then uses the `equals` method implemented by that class. If `equals` is not implemented in this class, then MATLAB uses an inherited `equals` method. If no intermediate ancestor classes define this method, MATLAB uses the `equals` method defined by the `java.lang.Object` class.

To change the way MATLAB compares members of a class, implement your own `equals` method in your class definition.

Changing the Effect of double, string, and char

You can change the output of the MATLAB `double`, `string`, and `char` functions by defining your own Java methods, `toDouble`, `toString`, and `toChar`. For more information, see “Convert to MATLAB Numeric Types” on page 3-66 and “Convert to MATLAB Strings” on page 3-67.

See Also

More About

- “Functions to Convert Java Objects to MATLAB Types” on page 3-66

How MATLAB Handles Undefined Methods

If your MATLAB command invokes a nonexistent method on a Java object, MATLAB looks for a function with the same name. If MATLAB finds a function of that name, it attempts to invoke it. If MATLAB does not find a function with that name, it displays a message stating that it cannot find a method by that name for the class.

For example, MATLAB has a function named `size`, and the Java API `java.awt.Frame` class also has a `size` method. If you call `size` on a `Frame` object, the `size` method defined by `java.awt.Frame` is executed. However, if you call `size` on an object of `java.lang.String`, MATLAB does not find a `size` method for this class. Therefore, it executes the MATLAB `size` function instead.

```
text = java.lang.String('hello');
size(text)

ans =
     1     1
```

Note When you define a Java class for use in MATLAB, avoid giving any of its methods the same name as a MATLAB function.

See Also

Avoid Calling Java main Methods in MATLAB

When calling a `main` method from MATLAB, the method returns when it executes its last statement, even if the method creates a thread that is still executing. In other environments, the `main` method does not return until the thread completes execution.

Be cautious when calling `main` methods from MATLAB, particularly `main` methods that start a user interface. `main` methods are written assuming they are the entry point to application code. When called from MATLAB, `main` is not the entry point, and the fact that other Java UI code might be already running can lead to problems.

See Also

Pass Data to Java Methods

MATLAB Type to Java Type Mapping

When you pass MATLAB data as arguments to Java methods, MATLAB converts the data into types that best represent the data to the Java language. For information about type mapping when passing data to arguments of type `java.lang`, see “Pass Java Objects” on page 3-61.

Each row in the following table shows a MATLAB type followed by the possible Java argument matches, from left to right in order of closeness of the match. The MATLAB types (except cell arrays) can be scalar (1-by-1) arrays or matrices. The Java types can be scalar values or arrays.

MATLAB Argument	Java Parameter Type (Scalar or Array) Types Other Than Object						
	Closest Type <-----> Least Close Type						
logical	boolean	byte	short	int	long	float	double
double	double	float	long	int	short	byte	boolean
single	float	double					
uint8 int8	byte	short	int	long	float	double	
uint16 int16	short	int	long	float	double		
uint32 int32	int	long	float	double			
uint64 int64	long	float	double				
string scalar, character vector, char scalar	String						

MATLAB Argument	Java Parameter Type (Scalar or Array) Types Other Than Object						
	Closest Type <-----> Least Close Type						
string array, cell array of character vectors See "Pass String Arguments" on page 3-60.	String[]						
Java object of type jClass	Java Object of type jClass	any superclass of jClass					
cell array of object	Object[]						
MATLAB object	Unsupported						

How Array Dimensions Affect Conversion

The term *dimension* means the number of subscripts required to address the elements of an array. For example, a 5-by-1 array has one dimension, because you index individual elements using one array subscript.

In converting MATLAB to Java arrays, MATLAB handles dimension in a special manner. For a MATLAB array, dimension is the number of nonsingleton dimensions in the array. For example, a 10-by-1 array has dimension 1. Whereas, a 1-by-1 array has dimension 0 because you cannot index into a scalar value. In Java code, the number of nested arrays determines the dimension. For example, `double[][]` has dimension 2, and `double` has dimension 0.

If the number of dimensions of the Java array matches the number of dimensions in MATLAB array `n`, then the converted Java array has `n` dimensions. If the Java array has fewer than `n` dimensions, then the conversion drops singleton dimensions, starting with the first one. The conversion stops when the number of remaining dimensions matches the number of dimensions in the Java array. If the Java array has more than `n` dimensions, then MATLAB adds trailing singleton dimensions.

Convert Numbers to Integer Arguments

When passing an integer type to a Java method that takes a Java integer parameter, the MATLAB conversion is the same as the Java conversion between integer types. In particular, if the integer is out-of-range, it does not fit into the number of bits of the parameter type. For out-of-range integers, MATLAB discards all lowest `n` bits. The value `n` is the number of bits in the parameter type. This conversion is unlike the conversion between MATLAB integer types, where out-of-range integers are converted to the maximum or minimum value represented by the destination type.

If the argument is a floating-point number, then MATLAB does not convert it to an integer in the same manner as Java. MATLAB first converts a floating-point number to a 64-bit signed integer with the fractional part truncated. Then the number is processed as if it were an `int64` argument.

A floating-point number is too large to be represented in a 64-bit integer when it is (outside the range from -2^{63} to 2^{63}). In which case, MATLAB uses the following conversions:

- `int`, `short`, and `byte` parameter values to `0`.
- `long` parameter values to `java.lang.Long.MIN_VALUE`.
- `Inf` and `-Inf` values to `-1`.
- `NaN` values to `0`.

Pass String Arguments

To call a Java method with an argument defined as `java.lang.String`, pass a MATLAB string or character vector. MATLAB converts the argument to a Java `String` object. You also can pass a `String` object returned by a Java method.

If the method argument is an array of type `String`, then pass a string array or a cell array of character vectors. MATLAB converts the input to a Java array of `String` objects, with dimensions adjusted as described in “How Array Dimensions Affect Conversion” on page 3-59.

Pass Java Objects

To call a method that has an argument belonging to a Java class (other than `java.lang.Object`), you must pass a Java object that is an instance of that class. MATLAB does not support Java autoboxing, the automatic conversion of MATLAB types to Java Object types. For example, MATLAB does not convert `double` to `java.lang.Double` for a parameter of type `Double`.

Pass `java.lang.Object`

A special case exists when the method takes an argument of the `java.lang.Object` class. Since this class is the root of the Java class hierarchy, you can pass objects of any class in the argument. MATLAB automatically converts the argument to the closest Java Object type, which might include Java-style autoboxing. This table shows the conversion.

MATLAB Argument	Java Object in Package <code>java.lang</code>
<code>logical</code>	<code>Boolean</code>
<code>double</code>	<code>Double</code>
<code>single</code>	<code>Float</code>
<code>char scalar</code>	<code>Character</code>
<code>string scalar</code> <code>nonempty char vector</code>	<code>String</code>
<code>uint8</code> <code>int8</code>	<code>Byte</code>
<code>uint16</code> <code>int16</code>	<code>Short</code>
<code>uint32</code> <code>int32</code>	<code>Integer</code>
<code>uint64</code> <code>int64</code>	<code>Long</code>
<code>string array (nonscalar)</code> <code>cell array of character vectors</code>	<code>String[]</code>
<code>Java object</code>	Argument unchanged
<code>cell array</code>	<code>Object[]</code>
<code>MATLAB object</code>	Unsupported

Pass Array of Objects

To call a method with an argument defined as `java.lang.Object` or an array of `java.lang.Object`, pass either a Java array or a MATLAB cell array. MATLAB automatically converts the cell array elements to their Java types as described in the “Pass `java.lang.Object`” on page 3-61 table. A Java array is an array returned from a Java constructor. You also can construct a Java array in MATLAB using the `javaArray` function.

Pass Cell Array of Java Objects

To create a cell array of Java objects, use the MATLAB syntax `{a1, a2, ...}`. You index into a cell array of Java objects in the usual way, with the syntax `a{m, n, ...}`. For example, create cell array `A`:

```
a1 = java.lang.Double(100);  
a2 = java.lang.Float(200);  
A = {a1,a2}
```

```
A =
```

```
1×2 cell array
```

```
 [1×1 java.lang.Double] [1×1 java.lang.Float]
```

Pass Empty Matrices, Nulls, and Missing Values

MATLAB converts an empty matrix as follows.

- If the argument is an empty character vector and the parameter is declared as `String`, then MATLAB passes in an empty (not `null`) Java `String` object.
- For all other cases, MATLAB converts an empty array to a Java `null`.

Empty (0-length) Java arrays remain unchanged.

MATLAB converts `<missing>` values in strings to `null`.

Overloaded Methods

When calling an overloaded method on a Java object, MATLAB compares the arguments you pass to the arguments defined for the methods. In this context, the term *method*

includes constructors. MATLAB determines the method to call and converts the arguments to Java types according to the Java conversion rules. For more information, see “Pass Array of Objects” on page 3-62.

When you call a Java method, MATLAB ensures that:

- 1 The object or class (for a static method) has a method by that name.
- 2 The invocation passes the same number of arguments of at least one method with that name.
- 3 Each passed argument is converted to the Java type defined for the method.

If all these conditions are satisfied, then MATLAB calls the method.

In a call to an overloaded method, if there is more than one candidate, MATLAB selects the one with arguments that best fit the calling arguments. First, MATLAB rejects methods that have argument types incompatible with the passed arguments. For example, if the method has a `double` argument, a `char` argument is incompatible.

MATLAB then selects the method with the highest fitness value, which is the sum of the fitness values of all its arguments. The fitness value for each argument is the fitness of the base type minus the difference between the MATLAB array dimension and the Java array dimension. For information about array dimensionality, see “How Array Dimensions Affect Conversion” on page 3-59. If two methods have the same fitness, then the first one defined in the Java class is chosen.

See Also

More About

- “Handle Data Returned from Java Methods” on page 3-64

Handle Data Returned from Java Methods

If a Java method returns a primitive data type, then MATLAB converts the data, as shown in the table in “Primitive Return Types” on page 3-64.

If a Java method signature specifies a return data of type `java.lang.Object`, then MATLAB converts the actual type returned, as shown in the table in “`java.lang.Object` Return Types” on page 3-65.

MATLAB does not convert other Java objects to MATLAB types. For information about handling this data, see “Functions to Convert Java Objects to MATLAB Types” on page 3-66.

Primitive Return Types

MATLAB converts primitive data returned from a Java method into types that best represent the data to the MATLAB language. This table shows how MATLAB converts the data. For some Java types, MATLAB treats scalar and array returns differently.

Java Return Type	Resulting MATLAB Type – Scalar	Resulting MATLAB Type – Array
<code>boolean</code>	<code>logical</code>	<code>logical</code>
<code>byte</code>	<code>double</code>	<code>int8</code>
<code>short</code>	<code>double</code>	<code>int16</code>
<code>int</code>	<code>double</code>	<code>int32</code>
<code>long</code>	<code>double</code>	<code>int64</code>
<code>float</code>	<code>double</code>	<code>single</code>
<code>double</code>	<code>double</code>	<code>double</code>
<code>char</code>	<code>char</code>	<code>char</code>

Example

The signature for the `java.lang.String` method `toCharArray` is:

```
public char[] toCharArray()
```

Call the method on a `String` object. MATLAB converts the output to a `char` array.


```
str = java.lang.String('hello');
res = str.toCharArray'

res =

    1x5 char array

hello
```

java.lang.Object Return Types

When a Java method is declared to return data of type `java.lang.Object`, MATLAB converts its value depending on the actual type returned. This table shows how MATLAB converts the data.

Actual Java Type	Resulting MATLAB Type — Scalar
<code>java.lang.Boolean</code>	logical
<code>java.lang.Byte</code>	double
<code>java.lang.Short</code>	double
<code>java.lang.Integer</code>	double
<code>java.lang.Long</code>	double
<code>java.lang.Float</code>	double
<code>java.lang.Double</code>	double
<code>java.lang.Character</code>	char
<code>java.lang.String</code>	char

There is no conversion if the return argument is a subclass of `Object` or an array of `Object`. The object remains a Java object. However, if you index into a returned `Object` array, MATLAB converts the value according to the table. For more information, see “Converting Object Array Elements to MATLAB Types” on page 3-29.

Example

Refer to the following signature for a `getData` method.

```
java.lang.Object getData()
```

If `getData` returns a `java.lang.Integer` object, then MATLAB converts the value to double.

Functions to Convert Java Objects to MATLAB Types

MATLAB only converts object data return values if the method signature specifies `java.lang.Object`. If the signature specifies any other object type, then MATLAB does not convert the value. For example, MATLAB does convert the return value for this method signature:

```
java.lang.Object getData()
```

But MATLAB does not convert the return value for this method:

```
java.lang.String getData()
```

To convert Java object data to MATLAB data, use MATLAB functions as described in these topics:

- “Convert to MATLAB Numeric Types” on page 3-66
- “Convert to MATLAB Strings” on page 3-67
- “Convert to MATLAB Structure” on page 3-67
- “Convert to MATLAB Cell Array” on page 3-67

Convert to MATLAB Numeric Types

To convert Java numeric types to MATLAB types, use a numeric MATLAB function like `double`. The action taken by the `double` function depends on the class of the object you specify.

- If the object is an instance of a class derived from `java.lang.Number`, then MATLAB converts the object to a MATLAB `double`.
- If the object is not an instance of a numeric class, then MATLAB checks the class definition for a `toDouble` method. MATLAB calls this method to perform the conversion.
- If you create your own class, then write a `toDouble` method to specify your own type conversion.

Note If the class of the object is not derived from `java.lang.Number` and it does not implement a `toDouble` method, then the `double` function displays an error message.

Convert to MATLAB Strings

To convert `java.lang.String` objects and arrays to MATLAB strings or character vectors, use the MATLAB `string` or `char` function.

If the object specified in the MATLAB function is not an instance of the `java.lang.String` class, then MATLAB checks its class definition for a `toString` or a `toChar` method. If you create your own class, then write a `toString` or `toChar` method to specify the string conversion.

Note If the class of the object is not `java.lang.String` and it does not implement a `toChar` method, then the `char` function displays an error message.

Convert to MATLAB Structure

If a Java class defines field names, then use the `struct` function to convert the object data to a MATLAB structure.

Suppose that you call a Java method that returns a `java.awt.Polygon` object. The class defines fields `xpoints` and `ypoints`. To run this example, create a `polygon` variable.

```
polygon = java.awt.Polygon([14 42 98 124],[55 12 -2 62],4);
```

Convert the object to a structure and display the x,y coordinates for the third point.

```
pstruct = struct(polygon)
```

```
pstruct =
```

```
    struct with fields:
```

```
    npoints: 4  
    xpoints: [4×1 int32]  
    ypoints: [4×1 int32]
```

Convert to MATLAB Cell Array

If your Java methods return different types of data, then use the `cell` function to convert the data to MATLAB types. Elements of the resulting cell array are converted according to the “Primitive Return Types” on page 3-64 and “`java.lang.Object` Return Types” on page 3-65 tables.

Suppose that you call Java methods that return arguments of type `java.lang.Double`, `java.awt.Point`, and `java.lang.String`. To run this example, create variables of these types.

```
import java.lang.* java.awt.*

% Create a Java array of double
dblArray = javaArray('java.lang.Double',1,10);
for m = 1:10
    dblArray(1,m) = Double(m * 7);
end

% Create a Java array of points
ptArray = javaArray('java.awt.Point',3);
ptArray(1) = Point(7.1,22);
ptArray(2) = Point(5.2,35);
ptArray(3) = Point(3.1,49);

% Create a Java array of strings
strArray = javaArray('java.lang.String',2,2);
strArray(1,1) = String('one');
strArray(1,2) = String('two');
strArray(2,1) = String('three');
strArray(2,2) = String('four');
```

Convert each array to a cell array. You can use `cellArray` in MATLAB functions.

```
cellArray = {cell(dblArray),cell(ptArray),cell(strArray)}
```

```
cellArray =
```

```
    1×3 cell array
```

```
    {1×10 cell}    {3×1 cell}    {2×2 cell}
```

Each cell holds an array of a different type. Display the contents.

```
cellArray{1,1}      % Array of type double
```

```
ans =
```

```
    1×10 cell array
```

```
    [7]    [14]    [21]    [28]    [35]    [42]    [49]    [56]    [63]    [70]
```

```
cellArray{1,2}      % Array of type Java.awt.Point
```

```
ans =  
  
    3×1 cell array  
  
    [1×1 java.awt.Point]  
    [1×1 java.awt.Point]  
    [1×1 java.awt.Point]  
  
cellArray{1,3}      % Array of type char array  
  
ans =  
  
    2×2 cell array  
  
    'one'      'two'  
    'three'    'four'
```

See Also

More About


- “Pass Data to Java Methods” on page 3-58
- “How MATLAB Represents Java Arrays” on page 3-21
- “Converting Object Array Elements to MATLAB Types” on page 3-29

Java Heap Memory Preferences

You can adjust the amount of memory that MATLAB allocates for Java objects.

Note The default heap size is sufficient for most cases.

To adjust the Java heap size:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**. Select **MATLAB > General > Java Heap Memory**.
- 2 Select a Java heap size value using the slider or spin box.

Note Increasing the Java heap size decreases the amount of memory available for storing data in arrays.

- 3 Click **OK**.
- 4 Restart MATLAB.

If the amount of memory you specified is not available upon restart, then MATLAB resets the value to the default, and displays an error dialog box. To readjust the value, repeat the previous steps.

If increasing the heap size does not eliminate memory errors, then check your Java code for memory leaks. Eliminate references to objects that are no longer useful. For more information, see the **Troubleshooting Java SE** guide.

Read and Write MATLAB MAT-Files in C/C++ and Fortran

- “Choosing Applications to Read and Write MATLAB MAT-Files” on page 4-2
- “Custom Applications to Access MAT-Files” on page 4-4
- “MAT-File API Library and Include Files” on page 4-7
- “What You Need to Build Custom Applications” on page 4-9
- “Copy External Data into MAT-File Format with Standalone Programs” on page 4-10
- “Create MAT-File in C or C++” on page 4-15
- “Read MAT-File in C/C++” on page 4-17
- “Create MAT-File in Fortran” on page 4-19
- “Read MAT-File in Fortran” on page 4-20
- “Work with mxArray” on page 4-21
- “Table of MAT-File Source Code Files” on page 4-24
- “Build on macOS and Linux Operating Systems” on page 4-26
- “Build on Windows Operating Systems” on page 4-29
- “Share MAT-File Applications” on page 4-30

Choosing Applications to Read and Write MATLAB MAT-Files

Under certain circumstances, you might need to write a custom program to interact with MATLAB MAT-file data. These programs are called applications to read MAT-files. Before writing a custom application to read MAT-file data, determine if MATLAB meets your data exchange needs by reviewing the following topics.

- The `save` and `load` functions.
- “Supported File Formats for Import and Export”.
- The `importdata` function and “Import Images, Audio, and Video Interactively”.
- “Methods for Importing Data”.

MATLAB supports applications written in C, C++, or Fortran to read MAT-files. To create the application, write your program using MATLAB APIs, then build using the `mex` command.

To write C/C++ applications, see:

- “MATLAB C API to Read MAT-File Data”
- “C Matrix API”
- C MEX API (optional)

To write Fortran applications, see:

- “MATLAB Fortran API to Read MAT-File Data”
- “Fortran Matrix API”
- “Fortran MEX API” (optional)

See Also

`importdata`

More About

- “Supported File Formats for Import and Export”
- “Methods for Importing Data”

- “Save and Load Workspace Variables”

Custom Applications to Access MAT-Files

In this section...

“Why Write Custom Applications?” on page 4-4
--

“MAT-File Interface Library” on page 4-5
--

“Exchanging Data Files Between Platforms” on page 4-6

Why Write Custom Applications?

To bring data into a MATLAB application, see “Methods for Importing Data”. To save data to a MAT-file, see “Save and Load Workspace Variables”. Use these procedures when you program your entire application in MATLAB, or if you share data with other MATLAB users. There are situations, however, when you must write a custom program to interact with data. For example:

- Your data has a custom format.
- You create applications for users who do not run MATLAB, and you want to provide them with MATLAB data.
- You want to read data from an external application, but you do not have access to the source code.

Before writing a custom application, determine if MATLAB meets your data exchange needs by reviewing the following topics.

- The save and load functions.
- “Supported File Formats for Import and Export”.
- The `importdata` function and “Import Images, Audio, and Video Interactively”.
- “Methods for Importing Data”.

If these features are not sufficient, you can create custom C/C++ or Fortran programs to read and write data files in the format required by your application. There are two types of custom programs:

- Standalone program — Run from a system prompt or execute in MATLAB (see “Run External Commands, Scripts, and Programs” on page 21-3). Requires MATLAB libraries to build the application.
- MEX file — Built and executed from the MATLAB command prompt. For information about creating and building MEX files, see “C MEX File Applications”.

MAT-File Interface Library

The MAT-File API contains routines for reading and writing MAT-files. Call these routines from your own C/C++ and Fortran programs. Use these routines, rather than attempt to write your own code, to perform these operations, since using the library insulates your applications from future changes to the MAT-file structure. For more information, see “MAT-File API Library and Include Files” on page 4-7.

MATLAB provides the `MATFile` type for representing a MAT-file.

MAT-File Routines

MAT-File API Function	Purpose
<code>matOpen</code>	Open a MAT-file.
<code>matClose</code>	Close a MAT-file.
<code>matGetDir</code>	Get a list of MATLAB arrays from a MAT-file.
<code>matGetVariable</code>	Read a MATLAB array from a MAT-file.
<code>matPutVariable</code>	Write a MATLAB array to a MAT-file.
<code>matGetNextVariable</code>	Read the next MATLAB array from a MAT-file.
<code>matDeleteVariable</code>	Remove a MATLAB array from a MAT-file.
<code>matPutVariableAsGlobal</code>	Put a MATLAB array into a MAT-file such that the load command places it into the global workspace.
<code>matGetVariableInfo</code>	Load a MATLAB array header from a MAT-file (no data).
<code>matGetNextVariableInfo</code>	Load the next MATLAB array header from a MAT-file (no data).

MAT-File C-Only Routines

<code>matGetFp</code>	Get an ANSI [®] C file pointer to a MAT-file.
-----------------------	--

The MAT-File Interface Library does not support MATLAB objects created by user-defined classes.

Do not create different MATLAB sessions on different threads using MAT-File API functions. MATLAB libraries are not multithread safe so you can use these functions only on a single thread at a time.

Exchanging Data Files Between Platforms

You can work with MATLAB software on different computer systems and send MATLAB applications to users on other systems. MATLAB applications consist of MATLAB code containing functions and scripts, and MAT-files containing binary data.

Both types of files can be transported directly between machines: MATLAB source files because they are platform independent, and MAT-files because they contain a machine signature in the file header. MATLAB checks the signature when it loads a file and, if a signature indicates that a file is foreign, performs the necessary conversion.

Using MATLAB across different machine architectures requires a facility for exchanging both binary and ASCII data between the machines. Examples of this type of facility include FTP, NFS, and Kermit. When using these programs, be careful to transmit MAT-files in *binary file mode* and MATLAB source files in *ASCII file mode*. Failure to set these modes correctly corrupts the data.

MAT-File API Library and Include Files

MATLAB provides include and library files to write programs to read and write MAT-files. The following table lists the path names to these files. The term *matlabroot* refers to the root folder of your MATLAB installation. The term *arch* is a unique string identifying the platform.

MAT-File API Folders

Platform	Contents	Folder
Microsoft® Windows	Include files	<i>matlabroot</i> \extern\include
	Libraries	<i>matlabroot</i> \bin\win64
	Examples	<i>matlabroot</i> \extern\examples\eng_mat
macOS Linux	Include files	<i>matlabroot</i> /extern/include
	Libraries	<i>matlabroot</i> /bin/ <i>arch</i>
	Examples	<i>matlabroot</i> /extern/examples/eng_mat

MAT-File API Include Files

The *matlabroot*\extern\include folder holds header files containing function declarations with prototypes for the routines that you can access in the API Library. These files are the same for Windows, macOS, and Linux systems. The folder contains:

- `mat.h` — Function prototypes for `mat` routines
- `matrix.h` — Definitions of the `mxArray` structure and function prototypes for matrix access routines

MAT-File API Libraries

The name of the libraries folder, which contains the shared (dynamically linkable) libraries, is platform-dependent.

Shared Libraries on Windows Systems

The `bin` folder contains the run-time version of the shared libraries:

- MAT-File library — *matlabroot*\extern\lib\win64\compiler\libmat.lib

- Matrix library — *matlabroot\extern\lib\win64\compiler\libmx.lib*
- MEX library (optional) — *matlabroot\extern\lib\win64\compiler\libmex.lib*

Shared Libraries on Linux Systems

The *bin/arch* folder, where *arch* is the value returned by the `computer('arch')` command, contains the shared libraries.

- MAT-File library — *matlabroot/bin/glnxa64/libmat.so*
- Matrix library — *matlabroot/bin/glnxa64/libmx.so*
- MEX library (optional) — *matlabroot/extern/bin/glnxa64/libmex.so*

Shared Libraries on macOS Systems

The *bin/arch* folder, where *arch* is the value returned by the `computer('arch')` command, contains the shared libraries. For example, on Apple macOS 64-bit systems, the folder is *bin/maci64*:

- MAT-File library — *matlabroot/bin/maci64/libmat.dylib*
- Matrix library — *matlabroot/bin/maci64/libmx.dylib*
- MEX library (optional) — *matlabroot/extern/bin/maci64/libmex.dylib*

Example Files

The *extern/examples/eng_mat* folder contains C/C++ and Fortran source code for examples demonstrating how to use the MAT-file routines.

What You Need to Build Custom Applications

To create a custom application, you need the tools and knowledge to modify and build source code. In particular, you need a compiler supported by MATLAB.

To exchange custom data with MATLAB data, use a MAT-file, a MATLAB format binary file. You do not need the MAT-file format specifications because the MAT-File Interface Library provides the API to the data. You need to know the details of your data to map it into MATLAB data. Get this information from your product documentation, then use the `mxArray` type in the Matrix Library to declare the data in your program.

In your custom program, use functions in the MATLAB C Matrix or Fortran Matrix API:

- MAT-File Interface Library
- Matrix Library

To build the application, use the `mex` build script with the `-client` engine option.

See Also

`mex` | `mxArray`

More About

- “MAT-File API Library and Include Files” on page 4-7
- “Build Engine Applications with IDE” on page 11-25

External Websites

- Supported and Compatible Compilers

Copy External Data into MAT-File Format with Standalone Programs

In this section...

- “Overview of `matimport.c` Example” on page 4-10
- “Declare Variables for External Data” on page 4-11
- “Create `mxArray` Variables” on page 4-11
- “Create MATLAB Variable Names” on page 4-12
- “Read External Data into `mxArray` Data” on page 4-12
- “Create and Open MAT-File” on page 4-13
- “Write `mxArray` Data to File” on page 4-13
- “Clean Up” on page 4-13
- “Build the Application” on page 4-13
- “Create the MAT-File” on page 4-13
- “Import Data into MATLAB” on page 4-14

Overview of `matimport.c` Example

This topic shows how to create a standalone program, `matimport`, to copy data from an external source into a MAT-file. The format of the data is custom, that is, it is not one of the file formats supported by MATLAB.

The `matimport.c` example:

- Creates variables to read the external data.
- Copies the data into `mxArray` variables.
- Assigns a variable name to each `mxArray`. Use these variable names in the MATLAB workspace.
- Writes the `mxArray` variables and associated variable names to the MAT-file.

To use the data in MATLAB:

- Build the standalone program `matimport`.
- Run `matimport` to create the MAT-file `matimport.mat`.

- Open MATLAB.
- Use one of the techniques described in “Save and Load Workspace Variables”.

The following topics describe these steps in detail. To see the code, open the file in the MATLAB Editor. The C statements in these topics are code snippets shown to illustrate a task. The statements in the topics are not necessarily sequential in the source file.

Declare Variables for External Data

There are two external data values, a string and an array of type `double`. The following table shows the relationship between the variables in this example.

External Data	Variable to Read External Data	mxArray Variable	MATLAB Variable Name
Array of type <code>double</code>	<code>extData</code>	<code>pVarNum</code>	<code>inputArray</code>
String	<code>extString</code>	<code>pVarChar</code>	<code>titleString</code>

The following statements declare the type and size for variables `extString` and `extData`.

```
#define BUFSIZE 256
char extString[BUFSIZE];
double extData[9];
```

Use these variables to read values from a file or a subroutine available from your product. This example uses initialization to create the external data.

```
const char *extString = "Data from External Device";
double extData[9] = { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 };
```

Create mxArray Variables

Functions in the MAT-File API use pointers of type `mxArray` to reference MATLAB data. These statements declare `pVarNum` and `pVarChar` as pointers to an array of any size or type.

```
/*Pointer to the mxArray to read variable extData */
mxArray *pVarNum;
/*Pointer to the mxArray to read variable extString */
mxArray *pVarChar;
```

To create a variable of the proper size and type, select one of the `mxCreate*` functions from the MX Matrix Library.

The size of `extData` is 9, which the example copies into a 3-by-3 matrix. Use the `mxCreateDoubleMatrix` function to create a two-dimensional, double-precision, floating-point `mxArray` initialized to 0.

```
pVarNum = mxCreateDoubleMatrix(3,3,mxREAL);
```

Use the `mxCreateString` function to create an `mxArray` variable for `extString`.

```
pVarChar = mxCreateString(extString);
```

Create MATLAB Variable Names

`matimport.c` assigns variable names `inputArray` and `titleString` to the `mxArray` data. Use these names in the MATLAB workspace. For more information, see “View Contents of MAT-File”.

```
const char *myDouble = "inputArray";  
const char *myString = "titleString";
```

Read External Data into mxArray Data

Copy data from the external source into each `mxArray`.

The C `memcpy` function copies blocks of memory. This function requires pointers to the variables `extData` and `pVarNum`. The pointer to `extData` is `(void *)extData`. To get a pointer to `pVarNum`, use one of the `mxGet*` functions from the Matrix API. Since the data contains only real values of type `double`, this example uses the `mxGetPr` function.

```
memcpy((void *) (mxGetPr(pVarNum)), (void *)extData, sizeof(extData));
```

The following statement initializes the `pVarChar` variable with the contents of `extString`.

```
pVarChar = mxCreateString(extString);
```

Variables `pVarNum` and `pVarChar` now contain the external data.

Create and Open MAT-File

The `matOpen` function creates a handle to a file of type `MATFile`. The following statements create a file pointer `pmat`, name the file `matimport.mat`, and open it for writing.

```
MATFile *pmat;  
const char *myFile = "matimport.mat";  
pmat = matOpen(myFile, "w");
```

Write mxArray Data to File

The `matPutVariable` function writes the `mxArray` and variable name into the file.

```
status = matPutVariable(pmat, myDouble, pVarNum);  
status = matPutVariable(pmat, myString, pVarChar);
```

Clean Up

To close the file:

```
matClose(pmat);
```

To free memory:

```
mxDestroyArray(pVarNum);  
mxDestroyArray(pVarChar);
```

Build the Application

To build the application, use the `mex` function with the `-client engine` option.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'matimport.c'), '.', 'f')  
mex -v -client engine matimport.c
```

Create the MAT-File

Run `matimport` to create the file `matimport.mat`. Either invoke the program from the system command prompt, or at the MATLAB command prompt, type:

```
!matimport
```

Import Data into MATLAB

Any user with a compatible version of MATLAB can read the `matimport.mat` file. Start MATLAB and use the `load` command to import the data into the workspace.

```
load matimport.mat
```

To display the variables, type:

```
whos
```

Name	Size	Bytes	Class
inputArray	3x3	72	double
titleString	1x43	86	char

See Also

Related Examples

- “Table of MAT-File Source Code Files” on page 4-24

Create MAT-File in C or C++

In this section...

“Create MAT-File in C” on page 4-15

“Create MAT-File in C++” on page 4-15

Create MAT-File in C

The `matcreat.c` example illustrates how to use the library routines to create a MAT-file that you can load into the MATLAB workspace. The program also demonstrates how to check the return values of MAT-file function calls for read or write failures. To see the code, open the file in MATLAB Editor.

After building the program, run the application. This program creates `mattest.mat`, a MAT-file that you can load into MATLAB. To run the application, depending on your platform, either double-click its icon or enter `matcreat` at the system prompt:

```
matcreat
```

```
Creating file mattest.mat...
```

To verify the MAT-file, at the MATLAB command prompt, type:

```
whos -file mattest.mat
```

Name	Size	Bytes	Class
GlobalDouble	3x3	72	double array (global)
LocalDouble	3x3	72	double array
LocalString	1x43	86	char array

```
Grand total is 61 elements using 230 bytes
```

Create MAT-File in C++

The C++ version of `matcreat.c` is `matcreat.cpp`. Open the file in MATLAB Editor.

See Also

Related Examples

- “Table of MAT-File Source Code Files” on page 4-24

Read MAT-File in C/C++

The `matdgn.c` example illustrates how to use the library routines to read and diagnose a MAT-file. To see the code, open the file in MATLAB Editor.

After building the program, run the application. This program reads the `mattest.mat` MAT-file created by the “Create MAT-File in C or C++” on page 4-15 example. To run the application, depending on your platform, either double-click its icon or enter `matdgn` at the system prompt.

```
matdgn mattest.mat
Reading file mattest.mat...
```

```
Directory of mattest.mat:
GlobalDouble
LocalString
LocalDouble
```

```
Examining the header for each variable:
According to its header, array GlobalDouble has 2 dimensions
    and was a global variable when saved
According to its header, array LocalString has 2 dimensions
    and was a local variable when saved
According to its header, array LocalDouble has 2 dimensions
    and was a local variable when saved
```

```
Reading in the actual array contents:
According to its contents, array GlobalDouble has 2 dimensions
    and was a global variable when saved
According to its contents, array LocalString has 2 dimensions
    and was a local variable when saved
According to its contents, array LocalDouble has 2 dimensions
    and was a local variable when saved
Done
```

See Also

Related Examples

- “Create MAT-File in Fortran” on page 4-19

- “Table of MAT-File Source Code Files” on page 4-24

Create MAT-File in Fortran

The `matdemo1.F` example creates the MAT-file, `matdemo.mat`. To see the code, open the file in MATLAB Editor.

After building the program, run the application. This program creates a MAT-file, `matdemo.mat`, that you can load into MATLAB. To run the application, depending on your platform, either double-click its icon or type `matdemo1` at the system prompt:

```
matdemo1
```

```
Creating MAT-file matdemo.mat ...
Done creating MAT-file
```

To verify the MAT-file, at the MATLAB command prompt, type:

```
whos -file matdemo.mat
```

Name	Size	Bytes	Class	Attributes
Numeric	3x3	72	double	
NumericGlobal	3x3	72	double	global
String	1x33	66	char	

Note For an example of a Microsoft Windows standalone program (not MAT-file specific), see `engwindemo.c` in the `matlabroot\extern\examples\eng_mat` folder.

See Also

Related Examples

- “Read MAT-File in C/C++” on page 4-17
- “Table of MAT-File Source Code Files” on page 4-24

Read MAT-File in Fortran

The `matdemo2.F` example illustrates how to use the library routines to read the MAT-file created by `matdemo1.F` and describe its contents. To see the code, open the file in MATLAB Editor.

After building the program, view the results.

```
matdemo2
```

```
Directory of Mat-file:  
String  
Numeric  
Getting full array contents:  
  1  
Retrieved String  
  With size  1-by- 33  
  3  
Retrieved Numeric  
  With size  3-by-  3
```

See Also

Related Examples

- “Table of MAT-File Source Code Files” on page 4-24

Work with mxArray

In this section...

“Read Structures from a MAT-File” on page 4-21

“Read Cell Arrays from a MAT-File” on page 4-22

The MAT-File Interface Library lets you access MATLAB arrays (type `mxArray`) in a MAT-file. To work directly with an `mxArray` in a C/C++ application, use functions in the Matrix Library.

You can find examples for working with the `mxArray` type in the `matlabroot/extern/examples/mex` and `matlabroot/extern/examples/mx` folders. The following topics show C code examples, based on these MEX examples, for working with cells and structures. The examples show how to read cell and structure arrays and display information based on the type of the `mxArray` within each array element.

If you create an application from one of the MEX examples, here are some tips for adapting the code to a standalone application.

- The MAT-file example, `matdgn.c`, shows how to open and read a MAT-file. For more information about the example, see “Read MAT-File in C/C++” on page 4-17.
- The MEX example, `explore.c`, has functions to read any MATLAB type using the `mxClassID` function. For more information about the example, see “Using Data Types” on page 7-11.
- Some MEX examples use functions, such as `mexPrintf`, from the C MEX API library, `libmex`. You do not need to use these functions to work with an `mxArray`, but if your program calls any of them, you must link to the MEX Library. To do this, add `libmex.lib` to the link statement.

Read Structures from a MAT-File

The `matreadstructarray.c` example is based on the `analyze_structure` function in `explore.c`. For simplicity, this example only processes real elements of type `double`; refer to the `explore.c` example for error checking and processing other types.

To see the code, open the file in the MATLAB Editor.

After building the program, run the application on the MAT-file, `testpatient.mat`.

First, create a structure `patient` and save it.

```
patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79 75 73; 180 178 177.5; 172 170 169];
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];

save testpatient.mat
```

Calculate the total of the `billing` field.

```
!matreadstruct testpatient.mat patient billing
```

```
Total for billing: 155.50
```

Read Cell Arrays from a MAT-File

The `matreadcellarray.c` example is based on the `analyze_cell` function in `explore.c`.

To see the code, open the file in the MATLAB Editor.

After building the program, run the application on the MAT-file, `testcells.mat`.

First, create three cell variables and save.

```
cellvar = {'hello'; [2 3 4 6 8 9]; [2; 4; 5]};
structvar = {'cell with a structure'; patient; [2; 4; 5]};
multicellvar = {'cell with a cell'; cellvar; patient};

save testcells.mat cellvar structvar multicellvar
```

Display the `mxArray` type for the contents of cell `cellvar`.

```
!matreadcell testcells.mat cellvar
```

```
0: string
1: numeric class
2: numeric class
```

See Also

Related Examples

- “Table of MAT-File Source Code Files” on page 4-24

Table of MAT-File Source Code Files

The `matlabroot/extern/examples/eng_mat` folder contains C/C++ and Fortran source code for examples demonstrating how to use the MAT-file routines. These examples create standalone programs. The source code is the same for both Windows, macOS, and Linux systems.

To build a code example, first copy the file to a writable folder, such as `c:\work` on your Windows path.

```
copyfile(fullfile(matlabroot,'extern','examples','eng_mat',...
'filename'), fullfile('c:','work'))
```

where *filename* is the name of the source code file.

For build information, see:

- “MAT-File API Library and Include Files” on page 4-7
- “Build on macOS and Linux Operating Systems” on page 4-26
- “Build on Windows Operating Systems” on page 4-29

Example	Description
<code>matcreat.c</code>	C program that demonstrates how to use the library routines to create a MAT-file that you can load into MATLAB.
<code>matcreat.cpp</code>	C++ version of the <code>matcreat.c</code> program.
<code>matdgns.c</code>	C program that demonstrates how to use the library routines to read and diagnose a MAT-file.
<code>matdemo1.F</code>	Fortran program that demonstrates how to call the MATLAB MAT-file functions from a Fortran program.
<code>matdemo2.F</code>	Fortran program that demonstrates how to use the library routines to read the MAT-file created by <code>matdemo1.F</code> and describe its contents.
<code>matimport.c</code>	C program based on <code>matcreat.c</code> used in the example for writing standalone applications.
<code>matreadstructarray.c</code>	C program based on <code>explore.c</code> to read contents of a structure array.

Example	Description
matreadcellarray.c	C program based on explore.c to read contents of a cell array.

For examples using the Matrix Library, see:

- “Tables of MEX Function Source Code Examples” on page 8-20.
- The explore.c example described in “Using Data Types” on page 7-11.

Build on macOS and Linux Operating Systems

In this section...

“Set Run-Time Library Path” on page 4-26

“Build Application” on page 4-27

Set Run-Time Library Path

At run time, you must tell the macOS and Linux operating system where the API shared libraries reside by setting an environment variable. The macOS or Linux command you use and the values you provide depend on your shell and system architecture. The following table lists the name of the environment variable (*envvar*) and the value (*pathspec*) to assign to it. The term *matlabroot* refers to the root folder of your MATLAB installation.

Operating System	<i>envvar</i>	<i>pathspec</i>
64-bit Apple Mac	DYLD_LIBRARY_PATH	<i>matlabroot</i> /bin/ maci64: <i>matlabroot</i> /sys/os/ maci64
64-bit Linux	LD_LIBRARY_PATH	<i>matlabroot</i> /bin/ glnxa64: <i>matlabroot</i> /sys/os/ glnxa64

Using the C Shell

Set the library path using the command.

```
setenv envvar pathspec
```

Replace the terms *envvar* and *pathspec* with the appropriate values from the table. For example, on a macOS system use:

```
setenv DYLD_LIBRARY_PATH  
matlabroot/bin/maci64:matlabroot/sys/os/maci64
```

You can place these commands in a startup script, such as `~/ .cshrc`.

Using the Bourne Shell

Set the library path using the command.

```
envvar = pathspec:envvar  
export envvar
```

Replace the terms *envvar* and *pathspec* with the appropriate values from the table. For example, on a macOS system use:

```
DYLD_LIBRARY_PATH=matlabroot/bin/maci64:matlabroot/sys/os/maci64:$DYLD_LIBRARY_PATH  
export DYLD_LIBRARY_PATH
```

You can place these commands in a startup script such as `~/ .profile`.

For more information, see `Append library path to "DYLD_LIBRARY_PATH" in MAC`.

Build Application

To compile and link MAT-file programs, use the `mex` script with the `-client engine` option.

This example shows how to build the example `matcreat.c`. Use this example to verify the build configuration for your system. `matcreat` is C program that demonstrates how to use the MAT-File API routines to create a MAT-file that you can load into MATLAB.

To build the example, first copy the source code to a writable folder on your path.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'matcreat.c'), '.', 'f')
```

Use the following command to build it.

```
mex -client engine matcreat.c
```

To modify the build instructions for your particular compiler, use the `-v -n` options to view the current compiler and linker settings. Then, modify the settings using the `mex varname=varvalue` option.

See Also

`mex`

Related Examples

- “Create MAT-File in C or C++” on page 4-15

Build on Windows Operating Systems

To compile and link MAT-file programs, use the `mex` script with the `-client engine` option.

This example shows how to build the example `matcreat.c`. Use this example to verify the build configuration for your system. `matcreat` is C program that demonstrates how to use the MAT-File API routines to create a MAT-file that you can load into MATLAB. For more information, see “Create MAT-File in C or C++” on page 4-15.

To build the example, first copy the source code to a writable folder on your path.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'matcreat.c'), '.', 'f')
```

Use the following command to build it.

```
mex -client engine matcreat.c
```

To modify the build instructions for your particular compiler, use the `-v -n` options to view the current compiler and linker settings. Then, modify the settings using the `mex varname=varvalue` option.

See Also

`mex`

Related Examples

- “Create MAT-File in C or C++” on page 4-15
- “Build Windows Engine Application” on page 11-13

Share MAT-File Applications

MATLAB requires shared library files for building a MAT-file application.

Library File Names by Operating System

Windows	Linux	Mac
libmat.dll	libmat.so	libmat.dylib
libmx.dll	libmx.so	libmx.dylib

If you share your MAT-file application with other users, then they must have an installed version of MATLAB. If you specify other run-time libraries in your build command, then these libraries must be present on the end-user's computer.

Calling Functions in C++ Shared Libraries from MATLAB

- “System Requirements” on page 5-2
- “Set Run-Time Library Path for C++ Interface” on page 5-3
- “Steps to Publish a MATLAB C++ Library Interface” on page 5-5
- “Publish Interface to Shared C++ Library on Windows” on page 5-7
- “Publish Interface to Shared C++ Library on Linux” on page 5-11
- “Call Functions in Windows Interface to C++ Shared Library” on page 5-13
- “Call Functions in Linux Interface to C++ Shared Library” on page 5-16
- “Publish Interface to Header-Only C++ Library” on page 5-19
- “Display Help for MATLAB Interface to C++ Library” on page 5-23
- “Publish Modified Help Text” on page 5-25
- “Define MATLAB Interface to C++ Library” on page 5-27
- “Sample C++ Library Definition File” on page 5-38
- “Call Functions in C++ Shared Library” on page 5-43
- “Limitations to C/C++ Support” on page 5-44
- “C++ Names That Are Invalid in MATLAB” on page 5-47
- “Troubleshooting C++ Interface Issues” on page 5-48
- “MATLAB to C++ Data Type Mapping” on page 5-49
- “Handling Exceptions” on page 5-53
- “Errors Parsing Header Files on macOS” on page 5-54
- “Build Error Due to Compile-Time Checks” on page 5-56

System Requirements

A shared library is a collection of classes and functions dynamically loaded by an application at run time. The MATLAB interface to a C++ shared library supports libraries containing functionality defined in C++ header files.

MATLAB supports dynamic libraries on all supported platforms.

Platform	Shared Library	File Extension
Microsoft Windows	Dynamic-link library file	.dll
Linux	Shared object file	.so
Apple macOS	Dynamic shared library	.dylib

Compiler Dependencies

Note Not every C++ compiler supports every C++ feature.

To build a MATLAB interface for a C++ library, you need an installed, MATLAB-supported C++ compiler. For an up-to-date list of supported compilers, see Supported and Compatible Compilers. You must build the interface library using the same compiler that was used to build the C++ library. If your library is header-only (does not use a `.lib` import library), then you can choose any supported C++ compiler to build the interface library.

See Also

More About

- “Steps to Publish a MATLAB C++ Library Interface” on page 5-5

External Websites

- Supported and Compatible Compilers

Set Run-Time Library Path for C++ Interface

If the C++ library has a shared library file, then that file and its dependencies must be on your system path or run-time search path (*rpath*). The publisher provides information about the library files. If the library is completely defined in header files, then there might not be a shared library file.

You can set the path each time that you work on the library or set it permanently by setting values in the system environment. To add the library to the system path permanently, refer to your operating system documentation. Do not use MATLAB commands to set the system path.

Temporarily Set Run-Time Library Path

To set the run-time library path *rtPath* temporarily, run one of the following commands before you start MATLAB.

- Windows Command Processor:

```
set PATH=rtPath;%PATH%
```

- Linux C shell:

```
setenv LD_LIBRARY_PATH rtPath
```

- Linux Bourne shell:

```
LD_LIBRARY_PATH=rtPath:LD_LIBRARY_PATH  
export LD_LIBRARY_PATH
```

- macOS: put the C++ library in the folder with the interface library or on the run-time search path.

Note If you use these commands, then you must set the path each time that you open the operating system prompt.

See Also

More About

- “Call Functions in C++ Shared Library” on page 5-43

External Websites

- Append library path to "DYLD_LIBRARY_PATH" in MAC

Steps to Publish a MATLAB C++ Library Interface

This topic outlines steps to take to publish a MATLAB interface to a C++ library. For examples, see:

- “Publish Interface to Shared C++ Library on Windows” on page 5-7
 - “Publish Interface to Header-Only C++ Library” on page 5-19
- 1** Review limitations on page 5-44 to MATLAB support for C++ libraries. This information tells you if it is possible to publish an interface to your library.
 - 2** Determine the functionality you want from the library.
 - You can use library examples as a guide.
 - Some libraries contain hundreds of classes and functions. To include all functionality in the interface, examine the constructors and methods in the definition file (see step 6) for missing information. By identifying specific functionality, you can limit the scope of the definition step.
 - 3** Identify and locate required header and library files, including dependencies.
 - Some libraries are written in C with a wrapper for C++. You need the dependent C header files and libraries to build the interface.
 - 4** Start MATLAB. Consider using the Live Script editor to execute the publishing commands and document the process. A Live Script:
 - Records the functionality excluded on page 5-52 from the interface.
 - Provides a place to document information outside the MATLAB environment, such as download information and system-level commands.
 - 5** Generate the `.mlx` definition file using `clibgen.generateLibraryDefinition`. Alternatively, call `clibgen.buildInterface` as described in “One-Step Build” on page 5-6.
 - 6** Examine the definition file for the functionality you determined in step 2. If necessary, define missing values on page 5-27 for this functionality.
 - 7** Build the interface using `build` on the library definition file. Be sure to add the interface file folder to the MATLAB path by clicking the link in the build message or by calling `addpath`.
 - 8** Test the MATLAB commands on page 5-43. Once you use a library class or function, you cannot modify the library definition unless you restart MATLAB and rebuild the library.

- 9 Package the MATLAB interface `.dll/.so/.dylib` library file for distribution, using “Create and Share Toolboxes”.

One-Step Build

If your library does not contain pointers or other constructs that require definition, you do not need to create a definition file. Call `clibgen.buildInterface` directly on the C++ header and library files. After building the interface, follow the steps to test the commands and package the interface.

See Also

`build` | `clibgen.buildInterface` | `clibgen.generateLibraryDefinition`

More About

- “Define MATLAB Interface to C++ Library” on page 5-27
- “Call Functions in C++ Shared Library” on page 5-43
- “Limitations to C/C++ Support” on page 5-44

Publish Interface to Shared C++ Library on Windows

This example uses the two-step build process, which generates a definition file for you to use to define certain inputs. The example shows how to define arguments specific to the library content. MATLAB provides C++ library and header files for you to use in this example. The library files are in this folder and its subfolders.

```
fullfile(matlabroot, 'extern', 'examples', 'cpp_interface')
```

Generate Definition File

This example calls `clibgen.generateLibraryDefinition` and guides you through defining inputs.

Create Publisher Folder

Create a folder for the MATLAB interface file.

```
pubPath = 'H:\Documents\MATLAB\publisher\matrixexample';
if ~isdir(pubPath)
    mkdir(pubPath)
end
cd(pubPath)
```

Identify C++ Library Files

Identify the names and paths to C++ library artifacts.

```
productPath = fullfile(matlabroot, 'extern', 'examples', 'cpp_interface');
libPath = fullfile(productPath, 'win64', 'mingw64');
% Header file name
hppFile = 'matrixOperations.hpp';
% Full path to folder containing all header files
hppPath = productPath;
% Full path to folder containing include files
iPath = hppPath;
% Library file name
libFile = 'matrixOperations.lib';
```

Name the Interface

```
myPkg = 'myPkg';
```

Call `clibgen.generateLibraryDefinition`

```
v = true;
clibgen.generateLibraryDefinition(fullfile(hppPath, hppFile), ...
```

```
'IncludePath', iPath, ...  
'Libraries', fullfile(libPath, libFile), ...  
'PackageName', myPkg, ...  
'Verbose', v)
```

Using MinGW64 Compiler (C++) compiler.
Generated definition file `definemyPkg.mlx` and data file `'myPkgData.xml'` contain definitions for 10 constructs supported by MATLAB.
5 constructs require additional definition. To include these constructs in the interface, edit the definitions in `definemyPkg.mlx`.
Build using `build(definemyPkg)`.

View Functionality

Although some constructs require additional definition, you can view the available functionality. If this functionality is sufficient for your needs, then you can continue with the “Build Library Interface” on page 5-9 step. Otherwise, continue with the step to define missing constructs.

```
summary(definemyPkg)
```

```
MATLAB Interface to myPkg1 Library
```

```
Class clib.myPkg1.Mat
```

```
Constructors:
```

```
clib.myPkg1.Mat()  
clib.myPkg1.Mat(clib.myPkg1.Mat)
```

```
Methods:
```

```
uint64 getLength()
```

```
No Properties defined
```

```
Functions
```

```
clib.myPkg1.updateMatByX(clib.myPkg1.Mat, int32)
```

Define Missing Constructs

To define the missing constructs, click the link in the `generateLibraryDefinition` output message to edit the definitions in `definemyPkg.mlx`. For information about editing this file and examples for specifying arguments, see “Define MATLAB Interface to C++ Library” on page 5-27.

- 1 Search the definition file for the `setMat` method and uncomment the statements defining it. To define the `src` argument, in this `defineArgument` statement, replace `<DIRECTION>` with `"input"` and `<SHAPE>` with `"len"`.

```
defineArgument(setMatDefinition, "src", "int32", "input", "len");
```

- 2 In the method `getMat`, define the `RetVal` output by replacing `<SHAPE>` with `"len"`.

```
defineOutput(getMatDefinition, "RetVal", "int32", "len");
```

- 3 In the method `copyMat`, define the `dest` argument by replacing `<DIRECTION>` with `"output"` and `<SHAPE>` with `"len"`.

```
defineArgument(copyMatDefinition, "dest", "int32", "output", "len");
```

- 4 In the function `addMat`, define the `mat` argument in function `addMat` by replacing `<SHAPE>` with `1`.

```
defineArgument(addMatDefinition, "mat", "clib.myPkg.Mat", "input", 1);
```

- 5 In the function `updateMatBySize`, define the `arr` argument by replacing `<DIRECTION>` with `"input"` and `<SHAPE>` with `"len"`.

```
defineArgument(updateMatBySizeDefinition, "arr", "int32", "input", "len");
```

Save and close the definition file.

Build Library Interface

Create the MATLAB interface file `myPkgInterface.dll`.

```
build(definemyPkg)
```

```
Building interface file 'myPkgInterface.dll'.
Interface file 'myPkgInterface.dll' built in folder
'H:\Documents\MATLAB\publisher\matrixexample\myPkg'.
To use the library, add the interface file folder to the MATLAB path.
```

Be sure to click the link in the message to add the interface file to the path.

Test the Interface

To test the interface, see “Call Functions in Windows Interface to C++ Shared Library” on page 5-13.

See Also

`clibgen.generateLibraryDefinition`

More About

- “Define MATLAB Interface to C++ Library” on page 5-27
- “Call Functions in Windows Interface to C++ Shared Library” on page 5-13

Publish Interface to Shared C++ Library on Linux

This example modifies the “Publish Interface to Shared C++ Library on Windows” on page 5-7 example to create a Linux version of the MATLAB interface.

Create a folder for the MATLAB interface file.

```
% Replace user with your user name
pubPath = '/mathworks/home/user/MATLAB/publisher/matrixexample/';
if ~isdir(pubPath)
    mkdir(pubPath)
end
cd(pubPath)
```

In the step Identify Path to C++ Library Files on page 5-0 , identify the Linux version of the library file and its location.

```
productPath = fullfile(matlabroot,'extern','examples','cpp_interface');
libPath = fullfile(productPath,'glnxa64');
% Header file name
hppFile = 'matrixOperations.hpp';
% Full path to folder containing all header files
hppPath = productPath;
% Full path to folder containing include files
iPath = hppPath;
% Library file name
libFile = 'libmwmatrixOperations.so';
```

Generate a definition file.

```
myPkg = 'myPkg';
v = true;
clibgen.generateLibraryDefinition(fullfile(hppPath,hppFile),...
'IncludePath', iPath,...
'Libraries', fullfile(libPath,libFile),...
'PackageName', myPkg,...
'Verbose',v)
```

To define missing constructs, follow the example steps in “Define Missing Constructs” on page 5-8.

Build the interface

```
build(definemyPkg)
```

To test the interface, see “Call Functions in Linux Interface to C++ Shared Library” on page 5-16.

See Also

More About

- “Publish Interface to Shared C++ Library on Windows” on page 5-7
- “Call Functions in Linux Interface to C++ Shared Library” on page 5-16

Call Functions in Windows Interface to C++ Shared Library

If you created the `matrixOperations` interface in the example “Publish Interface to Shared C++ Library on Windows” on page 5-7, then you can use it in the following example. The example assumes the `matrixOperations.dll` interface file is in this folder:

```
H:\Documents\MATLAB\publisher\matrixexample\myPkg
```

Set Paths

At the operating system prompt, add the path to the C++ shared library file. For more information, see “Set Run-Time Library Path for C++ Interface” on page 5-3.

```
set PATH=rtPath;%PATH%
```

where *rtPath* is the output of:

```
rtPath = fullfile(fullfile(matlabroot, 'extern', 'examples', 'cpp_interface'), 'win64', 'mingw64')
```

For example, type:

```
set PATH=C:\Program Files\MATLAB\R2019a\extern\examples\cpp_interface\win64\mingw64;%PATH%
```

Start MATLAB. Add the MATLAB interface file to the MATLAB path.

```
addpath('H:\Documents\MATLAB\publisher\matrixexample\myPkg')
```

View Help

At the MATLAB command prompt, display help for the interface. In the example, the `clibgen.generateLibraryDefinition` command changed the name of the interface to `matrixOperations` to `myPkg`. Type this command to load the package.

```
help clib.myPkg.Mat
```

To display the members of the package, type:

```
help clib.myPkg
```

```
Classes contained in clib.myPkg:
Mat          - clib.myPkg.Mat      Representation of C++ class Mat
```

```
Functions contained in clib.myPkg:  
addMat          - clib.myPkg.addMat      Representation of C++ function addMat  
updateMatByX    - clib.myPkg.updateMatByX  Representation of C++ function updateMatByX  
updateMatBySize - clib.myPkg.updateMatBySize Representation of C++ function updateMatBySize
```

To display signatures for the package function, click the links for `addMat`, `updateMatByX`, and `updateMatBySize`.

```
clib.myPkg.addMat      Representation of C++ function addMat  
  inputs  
  mat                 read-only clib.myPkg.Mat  
  outputs  
  RetVal             int32  
  
clib.myPkg.updateMatByX  Representation of C++ function updateMatByX  
  inputs  
  mat                 clib.myPkg.Mat  
  X                   int32  
  outputs  
  
clib.myPkg.updateMatBySize  Representation of C++ function updateMatBySize  
  inputs  
  mat                 clib.myPkg.Mat  
  arr                 int32  
  outputs
```

To display information about class `clib.myPkg.Mat`, click the link for `Mat`.

```
clib.myPkg.Mat      Representation of C++ class Mat  
Method Summary:  
Mat                - clib.myPkg.Mat      Constructor of C++ class Mat  
Mat                - clib.myPkg.Mat      Constructor of C++ class Mat  
setMat             - clib.myPkg.Mat.setMat  Method of C++ class Mat  
getMat             - clib.myPkg.Mat.getMat  Method of C++ class Mat  
getLength          - clib.myPkg.Mat.getLength  Method of C++ class Mat  
copyMat            - clib.myPkg.Mat.copyMat  Method of C++ class Mat
```

To display constructor and method signatures, use the `methods` or `methodsview` functions. For example, type:

```
methodsview(clib.myPkg.Mat)
```

Call Library Functions

Test the functions in the interface. For example, type:

```
matObj = clib.myPkg.Mat;    % Create a Mat object  
intArr = [1,2,3,4,5];  
matObj.setMat(intArr);     % Set the values to intArr  
retMat = matObj.getMat(5)  % Display the values  
  
retMat =
```

1x5 int32 row vector

1 2 3 4 5

See Also

More About

- “Publish Interface to Shared C++ Library on Windows” on page 5-7
- “Set Run-Time Library Path for C++ Interface” on page 5-3
- “Display Help for MATLAB Interface to C++ Library” on page 5-23

Call Functions in Linux Interface to C++ Shared Library

If you created the `matrixOperations` interface in the example “Publish Interface to Shared C++ Library on Linux” on page 5-11, then you can use it in the following example. The example assumes the `matrixOperations.dll` interface file is in folder with your user name:

```
/mathworks/home/user/MATLAB/publisher/matrixexample/myPkg
```

Set Paths

At the operating system prompt, add the path to the C++ shared library file. For more information, see “Set Run-Time Library Path for C++ Interface” on page 5-3.

```
setenv LD_LIBRARY_PATH rtPath
```

where *rtPath* is the output of:

```
rtPath = fullfile(fullfile(matlabroot, 'extern', 'examples', 'cpp_interface'), 'glnxa64')
```

Start MATLAB. Add the MATLAB interface file to the MATLAB path.

```
% Replace user with your user name  
addpath('/mathworks/home/user/MATLAB/publisher/matrixexample/myPkg')
```

View Help

At the MATLAB command prompt, display help for the interface. In the example, the `clibgen.generateLibraryDefinition` command changed the name of the interface to `matrixOperations` to `myPkg`. Type this command to load the package.

```
help clib.myPkg.Mat
```

To display the members of the package, type:

```
help clib.myPkg
```

```
Classes contained in clib.myPkg:  
Mat - clib.myPkg.Mat Representation of C++ class Mat  
  
Functions contained in clib.myPkg:  
addMat - clib.myPkg.addMat Representation of C++ function addMat  
updateMatByX - clib.myPkg.updateMatByX Representation of C++ function updateMatByX  
updateMatBySize - clib.myPkg.updateMatBySize Representation of C++ function updateMatBySize
```

To display signatures for the package function, click the links for `addMat`, `updateMatByX`, and `updateMatBySize`.

```

clib.myPkg.addMat      Representation of C++ function addMat
  inputs
  mat                  read-only clib.myPkg.Mat
  outputs
  RetVal              int32

clib.myPkg.updateMatByX  Representation of C++ function updateMatByX
  inputs
  mat                  clib.myPkg.Mat
  X                    int32
  outputs

clib.myPkg.updateMatBySize  Representation of C++ function updateMatBySize
  inputs
  mat                  clib.myPkg.Mat
  arr                  int32
  outputs

```

To display information about class `clib.myPkg.Mat`, click the link for `Mat`.

```

clib.myPkg.Mat      Representation of C++ class Mat
Method Summary:
  Mat                - clib.myPkg.Mat    Constructor of C++ class Mat
  Mat                - clib.myPkg.Mat    Constructor of C++ class Mat
  setMat             - clib.myPkg.Mat.setMat  Method of C++ class Mat
  getMat             - clib.myPkg.Mat.getMat  Method of C++ class Mat
  getLength         - clib.myPkg.Mat.getLength  Method of C++ class Mat
  copyMat           - clib.myPkg.Mat.copyMat  Method of C++ class Mat

```

To display constructor and method signatures, use the `methods` or `methodsview` functions. For example, type:

```
methodsview(clib.myPkg.Mat)
```

Call Library Functions

Test the functions in the interface. For example, type:

```

matObj = clib.myPkg.Mat;    % Create a Mat object
intArr = [1,2,3,4,5];
matObj.setMat(intArr);     % Set the values to intArr
retMat = matObj.getMat(5) % Display the values

retMat =

    1x5 int32 row vector

     1     2     3     4     5

```

See Also

More About

- “Publish Interface to Shared C++ Library on Linux” on page 5-11
- “Set Run-Time Library Path for C++ Interface” on page 5-3
- “Display Help for MATLAB Interface to C++ Library” on page 5-23

Publish Interface to Header-Only C++ Library

This example creates a MATLAB interface to a C++ library named `school`. The library is defined in the header file `school.hpp`.

This library defines classes of students and teachers by name and age. After you publish this library, MATLAB users can call functions in the `clib.school` package to create Student and Teacher objects.

Verify Selected C++ Compiler

Verify that you have selected a C++ compiler. For an up-to-date list of supported compilers, see Supported and Compatible Compilers. This example uses the MinGW64 compiler.

```
mex -setup cpp
```

Copy Example Header File

Copy the `school.hpp` header file to a writable folder on your path. This example assumes that the current folder is `H:\Documents\MATLAB\publisher`.

```
copyfile(fullfile(matlabroot,'extern','examples','cpp_interface','school.hpp'),'.','f')
```

Generate Definition File

```
clibgen.generateLibraryDefinition('school.hpp')
```

```
Warning: Only header files specified; assuming this is a header-only library.  
If library files also need to be specified, use the 'Libraries' option.
```

```
Using MinGW64 Compiler (C++) compiler.  
Generated definition file defineschool.mlx and data file 'schoolData.xml' contain definitions for  
21 constructs supported by MATLAB.  
1 constructs require additional definition. To include these constructs in the interface,  
edit the definitions in defineschool.mlx.  
Build using build(defineschool).
```

Define Missing Information for C++ Function getName

MATLAB reports that one construct requires additional information (definition). This means that MATLAB cannot automatically define the signature for one of the functions. To provide the missing information, click the link to open `defineschool.mlx` in MATLAB Live Editor.

Constructs with missing information are commented. Scroll through the file to locate the section titled *C++ function getName with MATLAB name clib.school.getName*. Uncomment the statements in the `getName` code section.

Modify this statement.

```
defineArgument(getNameDefinition, "p", "clib.school.Person", "input", <SHAPE>);
```

Replace `<SHAPE>` with the number 1 like this:

```
defineArgument(getNameDefinition, "p", "clib.school.Person", "input", 1);
```

Save the definition file.

For more information about including constructs in an interface, see “Define MATLAB Interface to C++ Library” on page 5-27.

Build Interface and Add to MATLAB Path

```
build(defineschool)
```

```
Building interface file 'schoolInterface.dll'.  
Interface file 'schoolInterface.dll' built in folder  
'H:\Documents\MATLAB\publisher\school'.  
To use the library, add the interface file folder to the MATLAB path.
```

Click the `interface` `file` `folder` link to add the interface to the path. Alternatively, type:

```
addpath('school')
```

View Contents of Library

```
summary(defineschool)
```

```
MATLAB Interface to school Library
```

```
Class clib.school.Person
```

```
Constructors:
```

```
clib.school.Person()  
clib.school.Person(string,uint64)  
clib.school.Person(clib.school.Person)
```

```
Methods:
```

```
setName(string)
```



```
setAge(uint64)
string getName()
uint64 getAge()
```

No Properties defined

Class `clib.school.Teacher`

Constructors:

```
clib.school.Teacher()
clib.school.Teacher(string,uint64)
clib.school.Teacher(clib.school.Teacher)
```

Methods:

```
string getName()
```

No Properties defined

Class `clib.school.Student`

Constructors:

```
clib.school.Student()
clib.school.Student(string,uint64)
clib.school.Student(clib.school.Student)
```

Methods:

```
string getName()
```

No Properties defined

Functions

```
string clib.school.getName(clib.school.Person)
```

Test Functions

To call functionality in the `school` interface, use the MATLAB `clib` package. Type:

```
clib.school.
```

then press **Tab**. The `school` library has three classes and one function.

Note Once you use a library class or function, you cannot modify the library definition unless you restart MATLAB and rebuild the library.

Create a teacher.

```
t1 = clib.school.Teacher('Ms. Jones',24);
```

Display teacher name.

```
getName(t1)
```

```
ans = "Ms. Jones"
```

Modify Help Text

You can provide help text for your users. For information, review the topic “Display Help for MATLAB Interface to C++ Library” on page 5-23.

Distribute Interface

To give the interface to a MATLAB user, instruct them to add the `schoolInterface.dll` file to a folder named `school` and add the folder to the MATLAB path. The package name is `clib.school`.

See Also

`build` | `clibgen.buildInterface` | `clibgen.generateLibraryDefinition`

More About

- “Define MATLAB Interface to C++ Library” on page 5-27
- “Display Help for MATLAB Interface to C++ Library” on page 5-23

Display Help for MATLAB Interface to C++ Library

Use the following MATLAB functions to view information about the members of an interface.

- `help` — View classes and functions in a package. When you publish an interface, you can add descriptive text. For more information, see “Publish Modified Help Text” on page 5-25.
- `methods` — View method names for a class.
- `methods` with `'-full'` option — View method signatures.
- `methodsview` — Table representation of method signatures. You might find the `methodsview` window easier to use as a reference guide because you do not need to scroll through the Command Window to find information.

If you created the `school` interface in the example “Publish Interface to Header-Only C++ Library” on page 5-19, then you can use it in the following examples. Assuming the `schoolInterface.dll` file is in the current folder, type:

```
addpath('.')
```

Display the classes and package functions.

```
help clib.school.
```

then press **Tab**. This command loads the package. MATLAB displays a list of members. Selecting the package, MATLAB displays:

```
Classes contained in clib.school:
Person          - clib.school.Person      Representation of C++ class Person
Teacher         - clib.school.Teacher     Representation of C++ class Teacher
Student         - clib.school.Student     Representation of C++ class Student

Functions contained in clib.school:
getName         - clib.school.getName     Representation of C++ function getName
```

To display the inputs and outputs for the `getName` package function, click the `getName` link.

```
clib.school.getName  Representation of C++ function getName
  inputs
    p                 clib.school.Person
  outputs
    RetVal            string
```

To display class methods, call the `methods` function for each class. For example, type:

```
methods clib.school.Person
```

```
Methods for class clib.school.Person:
```

```
Person eq ge getAge getName gt le lt ne setAge setName
```

```
Methods of clib.school.Person inherited from handle.
```

To display function signatures, call the `methodsview` function for each class. For example, type:

```
methodsview clib.school.Person
```

The function opens a window that displays the methods and information about arguments and returned values. For example, the signatures for the constructors are:

Return Type	Name	Arguments
clib.school.Person obj	Person	
clib.school.Person obj	Person	(name, uint64 scalar age)
clib.school.Person obj	Person	(clib.school.Person input1)

See Also

Publish Modified Help Text

When you publish an interface, the `clibgen.generateLibraryDefinition` function inserts default text about classes and functions. The `help` function displays this text to the user. You can modify the text by editing the definition file.

Note If you already loaded the `clib` package, for example, by calling `help` or calling a class constructor, then you must restart MATLAB to change the interface.

For example, the default help text for class `Person` is Representation of C++ class `Person`.

```
help clib.school
```

```
Classes contained in clib.school:
Person          - clib.school.Person    Representation of C++ class Person
Teacher         - clib.school.Teacher     Representation of C++ class Teacher
Student        - clib.school.Student     Representation of C++ class Student
```

To modify this text, edit `defineschool.mlx`. Search for the text Representation of C++ class `Person`.

Modify the "Description" value. Change:

```
"clib.school.Person    Representation of C++ class Person"
```

to:

```
"clib.school.Person    Class defined by name and age"
```

Save the file.

To rebuild the library, restart MATLAB. Navigate to the folder containing `defineschool.mlx`.

```
delete school\*.dll
build(defineschool)
addpath school
clib.school.Person
help clib.school
```

```
Classes contained in clib.school:
Person          - clib.school.Person    Class defined by name and age
Teacher         - clib.school.Teacher     Representation of C++ class Teacher
Student        - clib.school.Student     Representation of C++ class Student
```

See Also

`build | clibgen.generateLibraryDefinition`

More About

- “Define MATLAB Interface to C++ Library” on page 5-27

Define MATLAB Interface to C++ Library

After creating the definition file `defineLibName.mlx` using `clibgen.generateLibraryDefinition`, you might have to modify the contents in order to include functionality in the interface. Use the Live Editor to modify the file.

Note If library functionality is missing from the definition file, the library might contain unsupported language features or data types on page 5-44. For details, rerun `clibgen.generateLibraryDefinition` with the 'Verbose' option set to true.

Code Suggestions for Parameters

MATLAB offers code suggestions for values of the `DIRECTION` parameter on page 5-27, `SHAPE` parameter on page 5-28, and `MLTYPE` parameter on page 5-36. To activate suggestions for a specific parameter:

- Uncomment the code defining the function.
- Delete the parameter, including the `<>` characters.
- Pause to allow the code suggestions to display.
- If the suggestions do not appear, check that the `defineLibName.mlx` file is on your MATLAB path.

DIRECTION Parameter

In C++, pointer arguments can be used to pass and return data from a function. Use the `DIRECTION` parameter to specify if the argument is read-only input, output only, or a modifiable input argument.

The `DIRECTION` parameter has one of these values.

- `input` - Input argument only

If a pointer argument is used to pass data to the function, then it must appear as an input argument in the MATLAB signature.

- `output` - Output argument only

If a pointer argument is used to retrieve data from the function, then it must appear as an output argument in the MATLAB signature.

- `inputoutput` - Input and output argument

If a pointer argument is used to both pass and return data, then it must appear as both an input argument and an output argument.

For example, suppose that a C++ function `passData` has the following signature. The argument `data` might be an input to the function, the return value of the function, or input that the function modifies and returns. The documentation of the function tells you how the function uses the argument `data`.

```
void passData(double *data);
```

Note Default arguments with direction specified as `OUT` are not supported. Define these with `DIRECTION` as `INPUT` or `INPUTOUTPUT` in the MLX file.

Assuming `data` is a scalar double value, the following table shows the MATLAB signature based on its role.

C++ Signature	MATLAB Signature
<pre>// Pass data to C++ void passData(double *data);</pre>	<pre>% Set DIRECTION = input passData(data)</pre>
<pre>// Return data from C++ void passData(double *data);</pre>	<pre>% Set DIRECTION = output [data] = passData()</pre>
<pre>// Pass data to C++ and return the modified value void passData(double *data);</pre>	<pre>% Set DIRECTION = inputoutput [data] = passData(data)</pre>

SHAPE Parameter

In C++, pointer arguments are used for both scalar data and array data. To use a pointer as an array, dimension information is required to convert the array between C++ and MATLAB. The `SHAPE` parameter specifies the dimensions for the pointer.

Note These pointer types can only be used as scalars. Define `SHAPE` as `1` in the MLX file.

- Pointers representing arrays of C++ class objects
 - Pointers to non-const primitive arrays returned from a function
-

The following examples of constructs defined in the sample `cppUseCases.hpp` header file show you how to specify the shape of an argument. In these tables, the descriptions for the functions in the **C++ Signature** column are based on assumed knowledge of the arguments. The signature itself does not provide this information.

To view the `cppUseCases.hpp` header file and its generated definition file, see “Sample C++ Library Definition File” on page 5-38.

Define Pointer Argument to Fixed Scalar

C++ Signature	defineArgument Values
<p>The input to function <code>readScalarPtr</code> is a scalar pointer <code>in</code>.</p> <pre>void readScalarPtr(int const * in)</pre>	<p>For argument <code>in</code>, set SHAPE to 1.</p> <pre>defineArgument(readScalarPtrDefinition, "in", "int32", "input", 1);</pre>
<p>The input to function <code>readScalarPtr</code> is a scalar pointer to class <code>ns::MyClass2</code>.</p> <pre>void readScalarPtr(ns::MyClass2 const * in)</pre>	<p>For argument <code>in</code>, set SHAPE to 1.</p> <pre>defineArgument(readScalarPtrDefinition, "in", "clib.cppUseCases.ns.MyClass2", "input", 1);</pre>

Define Pointer Argument

C++ Signature	defineArgument Values
<p>The input to function <code>readMatrix1DPtr</code> is a pointer to an integer array of length <code>m</code>.</p> <pre>void readMatrix1DPtr(int const * mat, size_t m)</pre>	<p>For argument <code>mat</code>, set SHAPE to argument <code>m</code>.</p> <pre>defineArgument(readMatrix1DPtrDefinition, "mat", "int32", "input", "m");</pre>
<p>The input to function <code>readMatrix1DPtrFixedSize</code> is a pointer to a fixed-length array <code>mat</code>.</p> <pre>void readMatrix1DPtrFixedSize(int const * mat)</pre>	<p>For argument <code>mat</code>, set SHAPE to a fixed integer, such as 5.</p> <pre>defineArgument(readMatrix1DPtrFixedSizeDefinition, "mat", "int32", "input", 5);</pre>
<p>The input to function <code>readMatrix2DPtr</code> is a pointer to a two-dimensional integer matrix <code>mat</code> of size <code>m</code>-by-<code>n</code>.</p> <pre>void readMatrix2DPtr(int const * mat, size_t m, size_t n)</pre>	<p>For argument <code>mat</code>, set SHAPE to <code>["m", "n"]</code>.</p> <pre>defineArgument(readMatrix2DPtrDefinition, "mat", "int32", "input", ["m", "n"]);</pre>
<p>The input to function <code>readMatrix2DPtrFixedSize</code> is a pointer to a two-dimensional matrix <code>mat</code> of fixed dimensions.</p> <pre>void readMatrix2DPtrFixedSize(int const * mat)</pre>	<p>For argument <code>mat</code>, set SHAPE to a fixed integer, such as 6.</p> <pre>defineArgument(readMatrix2DPtrFixedSizeDefinition, "mat", "int32", "input", 6);</pre>

C++ Signature	defineArgument Values
<p>The input to function <code>readMatrix3DPtr</code> is a pointer to a three-dimensional matrix <code>mat</code> of size <code>m</code>-by-<code>n</code>-by-<code>p</code>.</p> <pre>void readMatrix3DPtr(int const * mat, size_t m,size_t n,size_t p)</pre>	<p>For argument <code>mat</code>, set SHAPE to ["m", "n", "p"].</p> <pre>defineArgument(readMatrix3DPtrDefinition, "mat", "int32", "input", ["m","n","p"]);</pre>

Define Array Argument

C++ Signature	defineArgument Values
<p>The input to function <code>readMatrix1DArr</code> is a one-dimensional array <code>mat</code> of length <code>len</code>.</p> <pre>void readMatrix1DArr(int const [] mat,size_t len)</pre>	<p>For argument <code>mat</code>, set SHAPE to length <code>len</code>.</p> <pre>defineArgument(readMatrix1DArrDefinition, "mat", "int32", "input", "len");</pre>

Define Output Pointer Argument

C++ Signature	defineArgument Values
<p>The input to function <code>getRandomValues</code> is a pointer to an array of length <code>len</code> which returns a pointer argument as output.</p> <pre>int const * getRandomValues(size_t len)</pre>	<p>For the return value <code>RetVal</code>, set SHAPE to argument 1.</p> <pre>defineOutput(getRandomValuesDefinition, "RetVal", "int32", "len");</pre>
<p>The output argument of function <code>getRandomValuesFixedSize</code> is a pointer to a fixed-length array.</p> <pre>int * getRandomValuesFixedSize()</pre>	<p>For the return value <code>RetVal</code>, set SHAPE to an integer, such as 5.</p> <pre>defineOutput(getRandomValuesFixedSizeDefinition, "RetVal", "int32", 5);</pre>

Define Scalar Object Argument

C++ Signature	defineArgument Values
<p>The input to function <code>addClassByPtr</code> is a pointer to class <code>ns::MyClass2</code>.</p> <pre>double addClassByPtr(ns::MyClass2 const * myc2)</pre>	<p>For the <code>myc2</code> argument, set SHAPE to 1.</p> <pre>defineArgument(addClassByPtrDefinition, "myc2", "cLib.cppUseCases.ns.MyClass2", "input", 1);</pre>
<p>The input to function <code>updateClassByPtr</code> is a pointer to class <code>ns::MyClass2</code>.</p> <pre>void updateClassByPtr(ns::MyClass2 * myc2, double a,short b,long c)</pre>	<p>For argument <code>myc2</code>, set SHAPE to 1.</p> <pre>defineArgument(updateClassByPtrDefinition, "myc2", "cLib.cppUseCases.ns.MyClass2", "input", 1);</pre>
<p>The input to function <code>readClassByPtr</code> is a pointer to class <code>ns::MyClass2</code>.</p> <pre>void readClassByPtr(ns::MyClass2 * myc2)</pre>	<p>For argument <code>myc2</code>, set SHAPE to 1.</p> <pre>defineArgument(readClassByPtrDefinition, "myc2", "cLib.cppUseCases.ns.MyClass2", "input", 1);</pre>
<p>The input to function <code>fillClassByPtr</code> is a pointer to class <code>ns::MyClass2</code>.</p> <pre>void fillClassByPtr(ns::MyClass2 * myc2, double a,short b,long c)</pre>	<p>For argument <code>myc2</code>, set SHAPE to 1.</p> <pre>defineArgument(fillClassByPtrDefinition, "myc2", "cLib.cppUseCases.ns.MyClass2", "input", 1);</pre>

Define Matrix Argument

C++ Signature	defineArgument Values
<p>The input to function <code>updateMatrix1DPtrByX</code> is a pointer to an integer vector of length <code>len</code>. The argument <code>x</code> modifies the input argument.</p> <pre>void updateMatrix1DPtrByX(int * mat,size_t len,int x)</pre>	<p>For argument <code>mat</code>, set <code>DIRECTION</code> to <code>inputoutput</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(updateMatrix1DPtrByXDefinition, "mat", "int32", "inputoutput", "len");</pre>
<p>The input to function <code>updateMatrix1DArrByX</code> is a reference to an integer array of length <code>len</code>. The input argument is modified by <code>x</code>.</p> <pre>void updateMatrix1DArrByX(int [] mat, size_t len,int x)</pre>	<p>For argument <code>mat</code>, set <code>DIRECTION</code> to <code>inputoutput</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(updateMatrix1DArrByXDefinition, "mat", "int32", "inputoutput", "len");</pre>
<p>The input to function <code>addValuesByPtr</code> is a pointer to an integer vector of length <code>len</code>. The function does not modify the input argument.</p> <pre>int addValuesByPtr(int * mat,size_t len)</pre>	<p>For argument <code>mat</code>, set <code>DIRECTION</code> to <code>input</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(addValuesByPtrDefinition, "mat", "int32", "input", "len");</pre>
<p>The input to function <code>addValuesByArr</code> is a reference to an integer array of length <code>len</code>. The function does not modify the input argument.</p> <pre>int addValuesByArr(int [] mat,size_t len)</pre>	<p>For argument <code>mat</code>, set <code>DIRECTION</code> to <code>input</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(addValuesByArrDefinition, "mat", "int32", "input", "len");</pre>

C++ Signature	defineArgument Values
<p>The function <code>fillRandomValuesToPtr</code> creates an integer vector of length <code>len</code> and returns a reference to the vector.</p> <pre>void fillRandomValuesToPtr(int * mat, size_t len)</pre>	<p>For argument <code>mat</code>, set <code>DIRECTION</code> to <code>output</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(fillRandomValuesToPtrDefinition, "mat", "int32", "output", "len");</pre>
<p>The function <code>fillRandomValuesToArr</code> creates an integer vector of length <code>len</code> and returns a reference to the vector.</p> <pre>void fillRandomValuesToArr(int [] mat, size_t len)</pre>	<p>For argument <code>mat</code>, set <code>DIRECTION</code> to <code>output</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(fillRandomValuesToArrDefinition, "mat", "int32", "output", "len");</pre>

Define String Arguments

C++ Signature	defineArgument Values
<p>The input to function <code>getStringCopy</code> is a null-terminated string.</p> <pre>char const * getStringCopy(char const * str)</pre>	<p>For argument <code>str</code>, set <code>MLTYPE</code> to <code>string</code> and <code>SHAPE</code> to <code>nullTerminated</code>.</p> <pre>defineArgument(getStringCopyDefinition, "str", "string", "input", "nullTerminated");</pre>
<p>The return value for function <code>getStringCopy</code> is a scalar of characters.</p> <pre>char const * getStringCopy(char const * str)</pre>	<p>For return value <code>RetVal</code>, set <code>MLTYPE</code> to <code>char</code> and <code>SHAPE</code> to <code>1</code>.</p> <pre>defineOutput(getStringCopy, "RetVal", "char", 1);</pre>
<p>The input to function <code>readCharArray</code> is a string specified by length <code>len</code>.</p> <pre>void readCharArray(char const * chArray, size_t len)</pre>	<p>For argument <code>chArray</code>, set <code>MLTYPE</code> to <code>char</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(readCharArrayDefinition, "chArray", "char", "input", "len");</pre>
<p>The input to function <code>readInt8Array</code> is an array of type <code>int8</code> and length <code>len</code>.</p> <pre>void readInt8Array(char const * int8Array, size_t len)</pre>	<p>For argument <code>int8Array</code>, set <code>MLTYPE</code> to <code>int8</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(readInt8ArrayDefinition, "int8Array", "int8", "input", "len");</pre>
<p>The return value for function <code>getRandomCharScalar</code> is a scalar of characters.</p> <pre>char const * getRandomCharScalar()</pre>	<p>For return value <code>RetVal</code>, set <code>MLTYPE</code> to <code>char</code> and <code>SHAPE</code> to <code>1</code>.</p> <pre>defineOutput(getRandomCharScalarDefinition, "RetVal", "char", 1);</pre>

C++ Signature	defineArgument Values
<p>The type of the return value for function <code>getRandomInt8Scalar</code> is <code>int8</code>.</p> <pre>char const * getRandomInt8Scalar()</pre>	<p>For return value <code>RetVal</code>, set <code>MLTYPE</code> to <code>int8</code> and <code>SHAPE</code> to <code>1</code>.</p> <pre>defineOutput(getRandomInt8ScalarDefinition, "RetVal", "int8", 1);</pre>
<p>The function <code>updateCharArray</code> updates the input argument <code>chArray</code>. The length of <code>chArray</code> is <code>len</code>.</p> <pre>void updateCharArray(char * chArray, size_t len)</pre>	<p>For argument <code>chArray</code>, set <code>DIRECTION</code> to <code>inputoutput</code> and <code>SHAPE</code> to <code>len</code>.</p> <pre>defineArgument(updateCharArrayDefinition, "chArray", "int8", "inputoutput", "len");</pre>

Define Typed Pointer Arguments

C++ Signature	defineArgument Values
<p>The input to function <code>useTypedefPtr</code> is a pointer to typedef <code>intDataPtr</code>.</p> <pre>void useTypedefPtr(intDataPtr input1)</pre> <p><code>intDataPtr</code> is defined as:</p> <pre>typedef int16_t intData; typedef intData * intDataPtr;</pre>	<p>For argument <code>input1</code>, set <code>DIRECTION</code> to <code>input</code> and <code>SHAPE</code> to <code>1</code>.</p> <pre>defineArgument(useTypedefPtrDefinition, "input1", "int16", "input", 1);</pre>

MLTYPE Parameter

MATLAB automatically converts C++ types to MATLAB types, as described in “MATLAB to C++ Data Type Mapping” on page 5-49. If the C++ type for the argument is a string, then use these options to choose values for the `MLTYPE` and `SHAPE` arguments.

C++ Type	MLTYPE	Options for SHAPE
char *	int8	Scalar value Array of scalar values
const char *	char	Scalar value Array of scalar values
	string	"nullTerminated"

Build Library and Review Contents

Build library *libName*.

```
build(definelibName)
addpath libName
```

Display the functionality in the library.

```
libDef = definelibName;
summary(libDef)
```

Note Saving LibraryDefinition object libDef into a MAT-file is not supported.

See Also

build | clibgen.LibraryDefinition

More About

- “Limitations to C/C++ Support” on page 5-44
- “Unsupported Data Types” on page 5-51

Sample C++ Library Definition File

This example shows a library definition file created from a C++ header file.

Example Header File

This example header file contains C++ language constructs that need further definition from a publisher in order to create a MATLAB interface to the library.

type `cppUseCases.hpp`

```
/*
 * cppUseCases.hpp
 *
 * Copyright 2019 The MathWorks, Inc.
 */

#ifndef cppUseCases_hpp
#define cppUseCases_hpp
#include <string>

// classes
// enums
// functions
// typedefs
// templates
// member functions
// constructors
// data members
// functions with pointers, reference for non fundamental types
// functions with pointers, reference, boxed Arr for fundamental types
// functions with const char *, char *, std::string
// explicit instantiation
// implicit instantiation
// functions with std::size_t, std::uint8_t inputs
// typedefs to value types and non-value types

namespace ns {

    class EmptyClass {};
    enum MyEnum1 { en1 = 1, en2, en4 = 4 };

    class MyClass1 {
```

```
public:
    int fundamentalProp;
    const int constProp = 5;
    bool boolProp;
    std::string strProp;
    EmptyClass nonFundamentalProp;
    MyEnum1 enumProp;

    MyClass1() {}
    MyClass1(const MyClass1& myc1) {
        this->fundamentalProp = myc1.fundamentalProp;
        this->boolProp = myc1.boolProp;
        this->strProp = myc1.strProp;
        this->nonFundamentalProp = myc1.nonFundamentalProp;
        this->enumProp = myc1.enumProp;
    }

    void setFundamentalProp(const int& fundProp);
};

class MyClass2 {
public:
    double a;
    short b;
    long c;
};

}

ns::MyEnum1 getMyEnum1Copy(ns::MyEnum1 en);

ns::MyClass1 getMyClass1Copy(ns::MyClass1 myc1);

// shape
double addClassByPtr(const ns::MyClass2* myc2);

double addClassByRef(const ns::MyClass2& myc2);

// direction

// inout
void updateClassByPtr(ns::MyClass2* myc2, double a, short b, long c);

void updateClassByRef(ns::MyClass2& myc2, double a, short b, long c);
```

```
// in
void readClassByPtr(ns::MyClass2* myc2);

void readClassByRef(ns::MyClass2& myc2);

// out
void fillClassByPtr(ns::MyClass2* myc2, double a, short b, long c);

void fillClassByRef(ns::MyClass2& myc2, double a, short b, long c);

// cases for shapes
void readScalarPtr(const int * in); // len = 1

void readScalarPtr(const ns::MyClass2 * in); // len = 1

void readMatrix1DPtr(const int * mat, size_t m);

void readMatrix1DPtrFixedSize(const int * mat); // fixed size = 5

void readMatrix2DPtr(const int * mat, size_t m, size_t n);

void readMatrix2DPtrFixedSize(const int * mat); // fixed size = 5

void readMatrix3DPtr(const int * mat, size_t m, size_t n, size_t p);

void readMatrix1DFixedArr(const int mat[5]);

void readMatrix2DFixedArr(const int mat[5][5]);

void readMatrix3DFixedArr(const int mat[5][5][5]);

void readMatrix1DArr(const int mat[], size_t len);

const int * getRandomValues(size_t len);

int * getRandomValuesFixedSize(); // fixed size = 5

// cases for direction

// inout
void updateMatrix1DPtrByX(int * mat, size_t len, int x);

void updateMatrix1DArrByX(int mat[], size_t len, int x);
```

```
void swap(int & a, int & b);

// in
int addValuesByPtr(int * mat, size_t len);
int addValuesByArr(int mat[], size_t len);
void readString(std::string & str);

// out
void fillRandomValuesToPtr(int * mat, size_t len);
void fillRandomValuesToArr(int mat[], size_t len);
void fillString(std::string & str);

// cases for type and shape
const char * getStringCopy(const char * str);
void readCharArray(const char * chArray, size_t len);
void readInt8Array(const char * int8Array, size_t len);
const char * getRandomCharScalar();
const char * getRandomInt8Scalar();

void updateCharArray(char * chArray, size_t len);

// template prototype
template <typename T>
class templClass {};

// explicit instantiation
template class templClass<int>;

// implicit instantiation
void useTemplClass(templClass<double> in) {}

// template with non-type parameter
template <int size>
class my_int_array {
public:
```

```
    int arr[size];
};

// explicit instantiation
template class my_int_array<4>;
void useTemplClassNonType(my_int_array<4> intArr4);

void useMixedTypes(std::size_t, std::int16_t, std::uint64_t);

// typedefs
typedef int16_t intData;
typedef intData * intDataPtr;

using clsData = ns::MyClass1;
using clsDataRef = clsData &;

void useTypedef(intData);
void useTypedefPtr(intDataPtr);
void useTypedefToClass(clsData);
void useTypedefRef(clsDataRef);
void useTypedefConstRef(const clsData &);

#endif
```

Library Definition File `definecppUseCases.mlx`

After generating the library definition file, replace `<SHAPE>`, `<DIRECTION>`, and `<MLTYPE>` with values so that MATLAB can create the appropriate function signatures. For details about providing this information, see “Define MATLAB Interface to C++ Library” on page 5-27.

```
edit definecppUseCases.mlx
```

Call Functions in C++ Shared Library

Set Path

Put the MATLAB interface file on the MATLAB path using the `addpath` function.

The C++ shared library file and its dependencies, if any, must be on your system path or run-time search path (`rpath`). For more information, see “Set Run-Time Library Path for C++ Interface” on page 5-3.

Call Function

To call a function in a C++ library, use the MATLAB `clib` package. MATLAB automatically loads the library when you type:

```
clib.fully-qualified-class-member
```

After MATLAB loads the library, you can use tab completion to view the members of the `clib` package.

For example, to call function `funcname` in library `libname`, type the following statement. `funcname` defines the input arguments `arg1`, `arg2`, ... and the output argument `retVal`.

```
retVal = clib.libname.funcname(arg1, arg2, ...)
```

See Also

More About

- “Set Run-Time Library Path for C++ Interface” on page 5-3
- “Display Help for MATLAB Interface to C++ Library” on page 5-23
- “MATLAB to C++ Data Type Mapping” on page 5-49

Limitations to C/C++ Support

You can create a MATLAB interface to shared libraries based on C++98 and commonly occurring C++11 features. However, if a library contains the following data types or language features, then the functionality is not included in the MATLAB interface to the library.

Data Types and Language Features Dropped From Interface

If a library contains the following data types or language features, then the functionality is not included in the MATLAB interface to the library.

When you build a library containing these features, MATLAB displays messages like:

```
Did not add member 'methodName' to class 'ClassName' at HeaderFile.h:290.  
'wchar_t' is not a supported type.
```

To view these messages, use the 'Verbose' option in the `clibgen.generateLibraryDefinition` or `clibgen.buildInterface` functions.

Data Types Dropped

- `wchar_t`
- `char16_t`
- `char32_t`
- Any type with a size greater than 64 bits, for example, `long double`
- `void *`
- `union`
- Pointers or arrays of `std::string`
- Pointers or references to enumerations
- Pointer, array, and reference data members
- `std` smart pointers other than `std::shared_ptr`
- Static data members
- Multilevel pointers, such as `type**` and `type***`
- Function pointer
- Class templates with incomplete or no instantiations

Language Features Detected and Dropped

- `move` constructor
- Overloaded operator functions
- Unnamed namespaces and classes
- Macro constants
- Global variables
- Variadic functions
- Function templates with incomplete or no instantiations

Language Features Not Supported

When you build a library containing these features, MATLAB displays:

Warning: Some C++ language constructs in the header file are not supported and not imported.

- Passing and returning Unicode® text from C++
- Creating objects of classes in `std` namespace, including custom classes
- Smart pointer semantics. Only `std::shared_ptr` is supported. MATLAB does not support operator overloads, move semantics, and the member functions in the class definition of the smart pointer which might restrict their effectiveness. Instead, a `std::shared_ptr<T>` behaves as type `T` in MATLAB. Type `T` can be passed as input for `std::shared_ptr<T>` and type `T` is received as output for `std::shared_ptr<T>`.
- Namespace aliases. Use original namespace name instead.

Unsupported Usage

- Passing `nullptr` to C++ functions or data members
- Certain class methods that have MATLAB names are not supported. These methods are `eq`, `ne`, `lt`, `gt`, `le`, and `ge`.

See Also

More About

- “MATLAB to C++ Data Type Mapping” on page 5-49

C++ Names That Are Invalid in MATLAB

MATLAB automatically renames classes, member functions, and namespaces with C++ names that are invalid in MATLAB using the `matlab.lang.makeValidName` function. For example, MATLAB converts the class name `_myclass` in library `mylib` to `x_myclass`. To use the class in MATLAB, type:

```
clib.mylib.x_myclass
```

When you publish an interface, you can modify `x_myclass` in the library definition file. For example, you can change the name to `myClass`. When you use the class in MATLAB, type:

```
clib.mylib.myClass
```

Renaming C++ namespaces or the MATLAB package is not supported.

Troubleshooting C++ Interface Issues

Why Is a Function Missing from the Definition File?

If a library contains any unsupported language features or data types on page 5-44, then the functionality is not included in the library definition file.

MATLAB Did Not Create .mlx Definition File

Sometimes, the content of library files exceeds limits which causes MATLAB to create an .m file instead of an .mlx file. Search your output folder for a definition file of the form `defineLibName.m` and edit the contents as you would the .mlx version.

Cannot Open .mlx Definition File

If you have issues opening the .mlx file, then you can use the corresponding `defineLibName.m` file instead. First delete the .mlx file, then open the .m file in MATLAB Editor.

See Also

More About

- “Limitations to C/C++ Support” on page 5-44
- “Unsupported Data Types” on page 5-51

MATLAB to C++ Data Type Mapping

The following table shows how MATLAB data types correspond to fixed-width C++ types.

Fixed-Width Data Types

C Exact-Width Integer Types	Equivalent MATLAB Type
int8_t	int8
uint8_t	uint8
int16_t	int16
uint16_t	uint16
int32_t	int32
uint32_t	uint32
int64_t	int64
uint64_t	uint64
C Floating Point Types	Equivalent MATLAB Type
float	single
double	double
Other C Types	Equivalent MATLAB Type
bool	logical

Non-Fixed-Width Integer Types

MATLAB maps non-fixed-width types to the corresponding fixed-width C types, based on the compiler used.

MATLAB supports the following non-fixed-width C integer types.

- short
- short int
- signed short
- signed short int

- unsigned short
- unsigned short int
- int
- signed int
- unsigned
- unsigned int
- long
- signed long
- signed long int
- unsigned long
- unsigned long int
- long long

String and Character Types

The following table shows how MATLAB converts string and char data returned from C++ into MATLAB types.

C++ String and Character Types

C++ Character Types	Equivalent MATLAB Type
char	int8 const char *
signed char	int8
unsigned char	uint8
C++ String Types	Equivalent MATLAB Type
std::string	string, ASCII characters only
const char *	string

User-Defined Types

C++ User-Defined Types	Equivalent MATLAB Type
struct <i>StructName</i>	<i>libname.StructName</i>
enum <i>EnumName</i>	<i>libname.EnumName</i>

nullptr Return Types

The C++ interface returns type-specific empty values for functions that return `nullptr`.

- For type `double`, MATLAB returns `[]` for the value `double.empty`.
- For all other fundamental types, MATLAB returns an `MLTYPE.empty` value. To determine `MLTYPE`, look for the C or C++ type in the tables in this topic. `MLTYPE` is in the **Equivalent MATLAB Types** column.

To test for `nullptr` types, call the `isempty` function.

- For non-fundamental types, MATLAB returns a `nullptr` object. To test for `nullptr` objects, call the `clibIsNull` function.

Unsupported Data Types

If the data type of an argument to a class constructor, method, or function is one of these types, or if the library contains any unsupported language features on page 5-44, then the functionality is not included in the MATLAB interface to the library.

- `wchar_t`
- `char16_t`
- `char32_t`
- Any type with a size greater than 64 bits, for example, `long double`
- `void *`
- `union`
- Pointers or arrays of `std::string`
- Pointers or references to enumerations
- Pointer, array, and reference data members
- `std` smart pointers other than `std::shared_ptr`

- Static data members
- Multilevel pointers, such as *type*** and *type****
- Function pointer
- Class templates with incomplete or no instantiations

Messages About Unsupported Types

MATLAB reports on constructs that use unsupported types. To view these messages, use the 'Verbose' option in the `clibgen.generateLibraryDefinition` or `clibgen.buildInterface` functions.

For example, suppose that `functionName` in `ClassName` is defined in `HeaderFile.h`. If an argument to `functionName` is of type `wchar_t`, then MATLAB does not add `functionName` to the definition file. In addition, if 'Verbose' is true, then `clibgen.generateLibraryDefinition` displays the following message.

```
Did not add member 'functionName' to class 'ClassName' at HeaderFile.h:290.  
'wchar_t' is not a supported type.
```

See Also

`clibgen.buildInterface` | `clibgen.generateLibraryDefinition`

Handling Exceptions

MATLAB supports `std::exception` and its subclasses. MATLAB catches the `std::exception` objects thrown from a library and displays text from the exception, if available.

Errors Parsing Header Files on macOS

The `clibgen.generateLibraryDefinition` and `clibgen.buildInterface` functions fail when parsing some header files on the macOS platform.

Suppose that you have two header files. Header file `simple1.hpp` includes a standard header such as `iostream` or `vector`. For example:

```
#ifndef SIMPLE1_HPP
#define SIMPLE1_HPP

#include <vector>

// class definitions
// functions
#endif
```

Header file `simple2.hpp` includes `simple1.hpp`. For example:

```
#include "simple1.hpp"

// class definitions based on simple1.hpp content
// other functionality
```

This call to `clibgen.generateLibraryDefinition` generates errors parsing header file on macOS.

```
clibgen.generateLibraryDefinition(["simple1.hpp", "simple2.hpp"], "PackageName", "simple")
```

To include this content in the library, create a wrapper header file with the contents of `simple1.hpp` and `simple2.hpp`. For example, create `wrapsimple.hpp` with these statements:

```
#ifndef SIMPLE1_HPP
#define SIMPLE1_HPP

#include <vector>

// class definitions
// functions
#endif

// Start of simple2.hpp content. Do not add the include simple1.hpp statement.
// class definitions based on simple1.hpp content
// other functionality
```

Create the library definition `definesimple.mlx`.

```
clibgen.generateLibraryDefinition("wrapsimple.hpp", "PackageName", "simple")
```

See Also

`clibgen.buildInterface` | `clibgen.generateLibraryDefinition`

Build Error Due to Compile-Time Checks

Compile-time assertions in a C++ shared library, such as `static_assert`, are not supported, but a construct with the assertion might be instantiated. To avoid an error when building a MATLAB interface to the library, comment the lines in the definition file that define the construct before building the interface.

See Also

`build`

More About

- “Steps to Publish a MATLAB C++ Library Interface” on page 5-5
- “Define MATLAB Interface to C++ Library” on page 5-27

Calling Functions in C Shared Libraries from MATLAB

- “Call C Functions in Shared Libraries” on page 6-2
- “Invoke Library Functions” on page 6-4
- “View Library Functions” on page 6-5
- “Load and Unload Library” on page 6-6
- “Limitations to Shared Library Support” on page 6-7
- “Limitations Using Structures” on page 6-11
- “Loading Library Errors” on page 6-13
- “No Matching Signature Error” on page 6-14
- “MATLAB Terminates Unexpectedly When Calling Function in Shared Library” on page 6-15
- “Pass Arguments to Shared C Library Functions” on page 6-16
- “Shared Library shrlibsample” on page 6-22
- “Pass String Arguments Examples” on page 6-23
- “Pass Structures Examples” on page 6-25
- “Pass Enumerated Types Examples” on page 6-31
- “Pass Pointers Examples” on page 6-33
- “Pass Arrays Examples” on page 6-38
- “Iterate Through an Array” on page 6-42
- “Pointer Arguments” on page 6-45
- “Structure Arguments” on page 6-48
- “Explore libstruct Objects” on page 6-50
- “MATLAB Prototype Files” on page 6-51

Call C Functions in Shared Libraries

A shared library is a collection of functions dynamically loaded by an application at run time. This MATLAB interface supports libraries containing functions defined in C header files. To call functions in C++ libraries, see the interface described in “C++ Libraries”.

MATLAB supports dynamic linking on all supported platforms.

Platform	Shared Library	File Extension
Microsoft Windows	dynamic link library file	.dll
Linux	shared object file	.so
Apple macOS	dynamic shared library	.dylib

A shared library needs a header file, which provides signatures for the functions in the library. A function signature, or prototype, establishes the name of the function and the number and types of its parameters. Specify the full path of the shared library and its header file.

You need an installed MATLAB-supported C compiler. For an up-to-date list of supported compilers, see Supported and Compatible Compilers.

MATLAB accesses C routines built into external, shared libraries through a command-line interface. This interface lets you load an external library into MATLAB memory and access functions in the library. Although types differ between the two language environments, usually you can pass types to the C functions without converting. MATLAB converts for you.

Details about using a shared library are in these topics.

- “Load and Unload Library” on page 6-6
- “View Library Functions” on page 6-5
- “Invoke Library Functions” on page 6-4

If the library function passes arguments, you need to determine the data type passed to and from the function. For information about data, see these topics.

- “Pass Arguments to Shared C Library Functions” on page 6-16
- “Manually Convert Data Passed to Functions” on page 6-20

- “Pointer Arguments” on page 6-45
- “Structure Arguments” on page 6-48

When you are finished working with the shared library, it is important to unload the library to free memory.

See Also

`calllib` | `libfunctions` | `loadlibrary`

More About

- “C and MATLAB Equivalent Types” on page 6-16
- “Limitations to Shared Library Support” on page 6-7

Invoke Library Functions

After loading a shared library into the MATLAB workspace, use the `calllib` function to call functions in the library. The syntax for `calllib` is:

```
calllib('libname', 'funcname', arg1, ..., argN)
```

Specify the library name, function name, and, if required, any arguments that get passed to the function.

See Also

More About

- “Pass Arguments to Shared C Library Functions” on page 6-16

View Library Functions

To display information about library functions in the MATLAB Command Window, use the `libfunctions` command.

To view function signatures, use the `-full` switch. This option shows the MATLAB syntax for calling functions written in C. The types used in the parameter lists and return values are MATLAB types, not C types. For more information on types, see “C and MATLAB Equivalent Types” on page 6-16.

To display information about library functions in a separate window, use the `libfunctionsview` function. MATLAB displays the following information:

Heading	Description
Return Type	Types the method returns
Name	Function name
Arguments	Valid types for input arguments

See Also

More About

- “C and MATLAB Equivalent Types” on page 6-16

Load and Unload Library

To give MATLAB access to functions in a shared library, first load the library into memory. After you load the library, you can request information about library functions and call them directly from the MATLAB command line.

To load a shared library into MATLAB, use the `loadlibrary` function. The most common syntax is:

```
loadlibrary('shrlib', 'hfile')
```

where *shrlib* is the shared library file name, and *hfile* is the name of the header file containing the function prototypes.

Note The header file provides signatures for the functions in the library and is a required argument for `loadlibrary`.

When you are finished working with the shared library, it is important to unload the library to free memory.

See Also

`loadlibrary` | `unloadlibrary`

Limitations to Shared Library Support

In this section...

“MATLAB Supports C Library Routines” on page 6-7

“Loading C++ Libraries” on page 6-7

“Limitations Using printf Function” on page 6-7

“Bit Fields” on page 6-8

“Enum Declarations” on page 6-8

“Unions Not Supported” on page 6-9

“Compiler Dependencies” on page 6-9

“Limitations Using Pointers” on page 6-9

“Functions with Variable Number of Input Arguments Not Supported” on page 6-10

MATLAB Supports C Library Routines

The MATLAB shared library interface supports C library routines only. Most professionally written libraries designed to be used by multiple languages and platforms work fine. For more information, see “Call C Functions in Shared Libraries” on page 6-2.

Many noncommercial libraries or libraries that have only been tested from C++ have interfaces that are not usable and require modification or an interface layer. In this case, we recommend using MEX files.

Loading C++ Libraries

The shared library interface does not support C++ classes or overloaded functions elements. Use the MATLAB C++ interface instead. For more information, see “C++ Libraries”.

Limitations Using printf Function

MATLAB does not display the output of the C `printf` function to the command window.

Bit Fields

You can modify a bit field declaration by using type `int` or an equivalent. For example, if your library has the following declared in its header file:

```
int myfunction();

struct mystructure
{
    /* note the sum of fields bits */
    unsigned field1 :4;
    unsigned field2 :4;
};
```

edit the header file and replace it with:

```
int myfunction();

struct mystructure
{
    /* field 8 bits wide to be manipulated in MATLAB */
    /* A char is 8 bits on all supported platforms */
    char allfields;
};
```

After editing the source code, rebuild the library. It is then possible to access the data in the two fields using bit masking in MATLAB.

Enum Declarations

`char` definitions for `enum` are not supported. In C, a `char` constant, for example `'A'`, is automatically converted to its numeric equivalent (65). MATLAB does not convert constants. To use this type of `enum`, edit the header file by replacing `'A'` with the number 65 (`int8('A') == 65`). For example, replace:

```
enum Enum1 {ValA='A',ValB='B'};
```

with:

```
enum Enum1 {ValA=65,ValB=66};
```

then rebuild the library.

Unions Not Supported

Unions are not supported. As a workaround, modify the source code taking out the union declaration and replacing it with the largest alternative. Then, to interpret the results, write MATLAB code as needed. For example, edit the source code and replace the following union:

```
struct mystruct
{
    union
    {
        struct {char byte1,byte2;};
        short word;
    };
};
```

with:

```
struct mystruct
{
    short word;
};
```

where on a little-endian based machine, `byte1` is `mod(f,256)`, `byte2` is `f/256`, and `word=byte2*256+byte1`. After editing the source code, rebuild the library.

Compiler Dependencies

Header files must be compatible with the supported compilers on a platform. For an up-to-date list of supported compilers, see [Supported and Compatible Compilers](#). You cannot load external libraries with explicit dependencies on other compilers.

Limitations Using Pointers

Function Pointers

The shared library interface does not support library functions that work with function pointers.

Multilevel Pointers

Limited support for multilevel pointers and structures containing pointers. Using inputs and outputs and structure members declared with more than two levels of indirection is

unsupported. For example, `double ***outp` translated to `doublePtrPtrPtr` is not supported.

Functions with Variable Number of Input Arguments Not Supported

The shared library interface does not support library functions with variable number of arguments, represented by an ellipsis (. . .).

You can create multiple alias functions in a prototype file, one for each set of arguments used to call the function. For more information, see “MATLAB Prototype Files” on page 6-51.

See Also

More About

- “Limitations Using Structures” on page 6-11

Limitations Using Structures

MATLAB Returns Pointers to Structures

MATLAB returns pointers to structures. Return by value is not supported.

Structure Cannot Contain Pointers to Other Structures

Nested structures or structures containing a pointer to a structure are not supported. However, MATLAB can access an array of structures created in an external library.

Requirements for MATLAB Structure Arguments

When you pass a MATLAB structure to an external library function, the field names must meet the following requirements.

- Every MATLAB field name must match a field name in the library structure definition.
- MATLAB structures cannot contain fields that are not in the library structure definition.
- If a MATLAB structure contains fewer fields than defined in the library structure, MATLAB sets undefined fields to zero.
- Field names are case-sensitive. For example, suppose that library `mylib` contains function `myfunc` with the following structure definition.

```
struct S {  
    double len;  
};
```

The field name is `len`. If you pass a structure to `myfunc` with the field name `Len`, MATLAB displays an error.

```
S.Len = 100;  
calllib('mylib', 'myfunc', S)
```

Requirements for C struct Field Names

When MATLAB loads a C `struct` definition, the field names in MATLAB are not case-sensitive. For example, when you load a library containing the following definition, MATLAB does not create two fields.

```
struct S {  
    double Num;  
    double num;  
};
```

See Also

More About

- “Limitations to Shared Library Support” on page 6-7

Loading Library Errors

Errors occur when the shared library is not a valid library. MATLAB displays messages similar to the following:

```
There was an error loading the library "F:\mylibs\testlib.dll"  
'F:\mylibs\testlib.dll' is not a valid shared library.
```

or

```
There was an error loading the library "/home/myname/testlib.so"  
'/home/myname/mylibs/testlib.so' has different architecture than the host.
```

If the library has dependencies which MATLAB cannot find, then MATLAB displays messages as described in “Invalid MEX File Errors” on page 7-71.

To find library dependencies on Windows systems, use the third-party product Dependency Walker. This free utility scans Windows modules and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions exported by that module, and which of those functions are called by other modules. See [How do I determine which libraries my MEX-file or stand-alone application requires?](#) for information on using the Dependency Walker.

No Matching Signature Error

This error occurs when you call a function without the correct input or output arguments, or if there is an error in the function signature in the header file.

For example, the function signature for the `addStructByRef` function in `shrlibsample` is:

```
[double, c_structPtr] addStructByRef(c_structPtr)
```

Load the library.

```
addpath(fullfile(matlabroot, 'extern', 'examples', 'shrlib'))  
loadlibrary('shrlibsample')
```

Create a structure, and call `addStructByRef`.

```
struct.p1 = 4;  
struct.p2 = 7.3;  
struct.p3 = -290;
```

If you call the function without the input argument, MATLAB displays the error message.

```
[res,st] = calllib('shrlibsample','addStructByRef')
```

```
Error using calllib  
No method with matching signature.
```

The correct call is:

```
[res,st] = calllib('shrlibsample','addStructByRef',struct)
```

See Also

[calllib](#) | [libfunctions](#)

MATLAB Terminates Unexpectedly When Calling Function in Shared Library

Some shared libraries, compiled as Microsoft Windows 32-bit libraries, use a calling convention that is incompatible with the default MATLAB calling convention. The default calling convention for MATLAB and for Microsoft C and C++ compilers is `cdecl`. For more information, see the MSDN® Calling Conventions article.

If your library uses a different calling convention, create a `loadlibrary` prototype file and modify it with the correct settings, as described in MATLAB Answers™ article Why does MATLAB crash when I make a function call on a DLL in MATLAB 7.6 (R2008a)?

See Also

`loadlibrary`

More About

- “MATLAB Prototype Files” on page 6-51

External Websites

- Calling Conventions

Pass Arguments to Shared C Library Functions

In this section...
“C and MATLAB Equivalent Types” on page 6-16
“How MATLAB Displays Function Signatures” on page 6-19
“NULL Pointer” on page 6-19
“Manually Convert Data Passed to Functions” on page 6-20

C and MATLAB Equivalent Types

The shared library interface supports all standard scalar C types. The following table shows these C types with their equivalent MATLAB types. MATLAB uses the type from the right column for arguments having the C type shown in the left column.

Note All scalar values returned by MATLAB are of type `double`.

MATLAB Primitive Types

C Type	Equivalent MATLAB Type
char, byte	int8
unsigned char, byte	uint8
short	int16
unsigned short	uint16
int	int32
long (Windows)	int32, long
long (Linux)	int64, long
unsigned int	uint32
unsigned long (Windows)	uint32, long
unsigned long (Linux)	uint64, long
float	single
double	double
char *	char array (1xn)
*char[]	cell array of character vectors

The following table shows how MATLAB maps C pointers (column 1) to the equivalent MATLAB function signature (column 2). Usually, you can pass a variable from the Equivalent MATLAB Type column to functions with the corresponding Argument Data Type. See “Pointer Arguments in C Functions” on page 6-45 for information about when to use a `lib.pointer` object instead.

MATLAB Extended Types

C Pointer Type	Argument Data Type	Equivalent MATLAB Type	Example Function in “Shared Library shrlibsample” on page 6-22
<code>double *</code>	<code>doublePtr</code>	<code>double</code>	<code>addDoubleRef</code>
<code>float *</code>	<code>singlePtr</code>	<code>single</code>	
<code>intsize * (integer pointer types)</code>	<code>(u)int(size)Ptr</code> For example, <code>int64 *</code> becomes <code>int64Ptr</code> .	<code>(u)int(size)</code>	<code>multiplyShort</code>
<code>byte[]</code>	<code>int8Ptr</code>	<code>int8</code>	
<code>char[]</code> (null-terminated string passed by value)	<code>cstring</code>	<code>char array (1xn)</code>	<code>stringToUpper</code>
<code>char **</code> (array of pointers to strings)	<code>stringPtrPtr</code>	cell array of character vectors	
<code>enum</code>	<code>enumPtr</code>		
<code>type **</code>	<code>typePtrPtr</code> For example, <code>double **</code> becomes <code>doublePtrPtr</code> .	<code>lib.pointer object</code>	<code>allocateStruct</code>
<code>void *</code>	<code>voidPtr</code>		<code>deallocateStruct</code>
<code>void **</code>	<code>voidPtrPtr</code>	<code>lib.pointer object</code>	
<code>struct</code> (C-style structure)	<code>structure</code>	<code>MATLAB struct</code>	<code>addStructFields</code>
<code>mxArray *</code>	<code>MATLAB array</code>	<code>MATLAB array</code>	
<code>mxArray **</code>	<code>MATLAB arrayPtr</code>	<code>lib.pointer object</code>	

How MATLAB Displays Function Signatures

Here are things to note about the input and output arguments shown in MATLAB function signatures.

- Many arguments (like `int32` and `double`) are similar to their C counterparts. In these cases, pass in the MATLAB types shown for these arguments.
- Some C arguments (for example, `**double`, or predefined structures), are different from standard MATLAB types. In these cases, either pass a standard MATLAB type and let MATLAB convert it for you, or convert the data yourself using the MATLAB functions `libstruct` and `libpointer`. For more information, see “Manually Convert Data Passed to Functions” on page 6-20.
- C functions often return data in input arguments passed by reference. MATLAB creates additional output arguments to return these values. Input arguments ending in `Ptr` or `PtrPtr` are also listed as outputs.

For an example of MATLAB function signatures, see “Shared Library `shrlibsample`” on page 6-22.

Guidelines for Passing Arguments

- Nonscalar arguments must be declared as passed by reference in the library functions.
- If the library function uses single subscript indexing to reference a two-dimensional matrix, keep in mind that C programs process matrices row by row. MATLAB processes matrices by column. To get C behavior from the function, transpose the input matrix before calling the function, and then transpose the function output.
- Use an empty array, `[]`, to pass a NULL parameter to a library function that supports optional input arguments. This notation is valid only when the argument is declared as a `Ptr` or `PtrPtr` as shown by `libfunctions` or `libfunctionsview`.

NULL Pointer

You can create a NULL pointer to pass to library functions in the following ways:

- Pass an empty array `[]` as the argument.
- Use the `libpointer` function:


```
p = libpointer; % no arguments
p = libpointer('string') % string argument
```

```
p = libpointer('cstring') % pointer to a string argument
```

- Use the `libstruct` function:

```
p = libstruct('structtype'); % structure type
```

Empty `libstruct` Object

To create an empty `libstruct` object, call `libstruct` with only the `structtype` argument. For example:

```
sci = libstruct('c_struct')  
get(sci)
```

```
p1: 0  
p2: 0  
p3: 0
```

MATLAB displays the initialized values.

Manually Convert Data Passed to Functions

Under most conditions, MATLAB software automatically converts data passed to and from external library functions to the type expected by the external function. However, you might choose to convert your argument data manually. For example:

- When passing the same data to a series of library functions, convert it once manually before calling the first function rather than having MATLAB convert it automatically on every call. This strategy reduces the number of unnecessary copy and conversion operations.
- When passing large structures, save memory by creating MATLAB structures that match the shape of the C structures used in the function instead of using generic MATLAB structures. The `libstruct` function creates a MATLAB structure modeled from a C structure taken from the library.
- When an argument to an external function uses more than one level of referencing (for example, `double **`), pass a pointer created using the `libpointer` function rather than relying on MATLAB to convert the type automatically.

See Also

`libfunctions` | `libfunctionsview` | `libpointer` | `libstruct`

Related Examples

- “Shared Library shrlibsample” on page 6-22

More About

- “Structure Arguments” on page 6-48

Shared Library `shrlibsample`

MATLAB includes a sample external library called `shrlibsample`. The library is in the folder `matlabroot\extern\examples\shrlib`.

View the source code in MATLAB.

```
edit([matlabroot '/extern/examples/shrlib/shrlibsample.c'])
edit([matlabroot '/extern/examples/shrlib/shrlibsample.h'])
```

To use the `shrlibsample` library, choose one of the following.

- Add the folder to your MATLAB path:

```
addpath(fullfile(matlabroot, 'extern', 'examples', 'shrlib'))
```

- Make the folder your current working folder:

```
cd(fullfile(matlabroot, 'extern', 'examples', 'shrlib'))
```

Load the library and display the MATLAB signatures for the functions in the library.

```
loadlibrary('shrlibsample')
libfunctions shrlibsample -full
```

Functions in library `shrlibsample`:

```
[double, doublePtr] addDoubleRef(double, doublePtr, double)
double addMixedTypes(int16, int32, double)
[double, c_structPtr] addStructByRef(c_structPtr)
double addStructFields(c_struct)
c_structPtrPtr allocateStruct(c_structPtrPtr)
voidPtr deallocateStruct(voidPtr)
lib.pointer exportedDoubleValue
lib.pointer getListOfStrings
doublePtr multDoubleArray(doublePtr, int32)
[lib.pointer, doublePtr] multDoubleRef(doublePtr)
int16Ptr multiplyShort(int16Ptr, int32)
doublePtr print2darray(doublePtr, int32)
printExportedDoubleValue
cstring readEnum(Enum1)
[cstring, cstring] stringToUpper(cstring)
```

Pass String Arguments Examples

In this section...

“stringToUpper Function” on page 6-23

“Convert MATLAB Character Array to Uppercase” on page 6-23

stringToUpper Function

The `stringToUpper` function in the `shrlibsample` library converts the characters in the input argument to uppercase. The input parameter, `char *`, is a C pointer to a string.

```
EXPORTED_FUNCTION char* stringToUpper(char *input)
{
    char *p = input;

    if (p != NULL)
        while (*p!=0)
            *p++ = toupper(*p);
    return input;
}
```

The function signature for `stringToUpper` is shown in the following table. MATLAB maps the C pointer type (`char *`) into `cstring` so you can pass a MATLAB character array to the function.

Return Type	Name	Arguments
[<code>cstring</code> , <code>cstring</code>]	<code>stringToUpper</code>	(<code>cstring</code>)

Convert MATLAB Character Array to Uppercase

This example shows how to pass a MATLAB character array `str` to a C function, `stringToUpper`.

```
str = 'This was a Mixed Case string';
```

Load the library containing the `stringToUpper` function.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

```
    loadlibrary('shrlibsample')  
end
```

Pass `str` to the function.

```
res = calllib('shrlibsample','stringToUpper',str)
```

```
res =  
'THIS WAS A MIXED CASE STRING'
```

The input parameter is a pointer to type `char`. However, a MATLAB character array is not a pointer, so the `stringToUpper` function does not modify the input argument, `str`.

```
str
```

```
str =  
'This was a Mixed Case string'
```

See Also

Related Examples

- “Shared Library `shrlibsample`” on page 6-22
- “Iterate Through an Array” on page 6-42

Pass Structures Examples

In this section...

“addStructFields and addStructByRef Functions” on page 6-25

“Add Values of Fields in Structure” on page 6-26

“Preconvert MATLAB Structure Before Adding Values” on page 6-27

“Autoconvert Structure Arguments” on page 6-28

“Pass Pointer to Structure” on page 6-28

addStructFields and addStructByRef Functions

The `shrlibsample` example library contains two functions with `c_struct` structure input parameters. `c_struct` is defined in the `shrlibsample.h` header file.

```
struct c_struct {
    double p1;
    short p2;
    long p3;
};
```

Both functions sum the values of the fields in the structure. The input to `addStructFields` is `c_struct`. The input to `addStructByRef` is a pointer to `c_struct`. This function also modifies the fields after summing the values.

addStructFields Function

The `addStructFields` function sums the values of the fields in a `c_struct` structure.

```
EXPORTED_FUNCTION double addStructFields(struct c_struct st)
{
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

The MATLAB function signature is:

Return Type	Name	Arguments
double	addStructFields	(struct c_struct)

addStructByRef Function

The `addStructByRef` function sums the values of the fields in a `c_struct` structure, then modifies the fields. The function returns the sum calculated before modifying the fields.

```
EXPORTED_FUNCTION double addStructByRef(struct c_struct *st) {
    double t = st->p1 + st->p2 + st->p3;
    st->p1 = 5.5;
    st->p2 = 1234;
    st->p3 = 12345678;
    return t;
}
```

Since the function modifies the input argument, MATLAB also returns the input as an output argument of type `c_structPtr`. The MATLAB function signature is:

Return Type	Name	Arguments
[double, c_structPtr]	addStructByRef	(c_structPtr)

You can pass a MATLAB structure to the function and let MATLAB autoconvert the argument. Or you can pass a pointer to a structure, which avoids creating a copy of the structure.

Add Values of Fields in Structure

This example shows how to pass a MATLAB structure to the function, `addStructFields`.

Create and initialize structure `sm`. Each field is of type `double`.

```
sm.p1 = 476;
sm.p2 = -299;
sm.p3 = 1000;
```

Load the library containing the `addStructFields` function.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Call the function. MATLAB automatically converts the fields of structure `sm` to the library definition for `c_struct`.

```
calllib('shrlibsample','addStructFields',sm)
ans = 1177
```

Preconvert MATLAB Structure Before Adding Values

This example shows how to preconvert structure `sm` to `c_struct` before calling `addStructFields`. If you repeatedly pass `sm` to functions, preconverting eliminates the processing time required by MATLAB to autoconvert the structure for each function call.

Create and initialize a MATLAB structure.

```
sm.p1 = 476;
sm.p2 = -299;
sm.p3 = 1000;
```

Load the library containing the `addStructFields` function.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Convert the fields, which are of type double, to match the `c_struct` structure types, double, short, and long.

```
sc = libstruct('c_struct',sm);
```

Display the field names and values.

```
get(sc)
    p1: 476
    p2: -299
    p3: 1000
```

Add the field values.

```
calllib('shrlibsample','addStructFields',sc)
ans = 1177
```

Autoconvert Structure Arguments

This example shows how to pass a MATLAB structure to a C library function, `addStructByRef`. When you pass the structure, MATLAB automatically converts the field types, but MATLAB also makes a copy of the fields.

Load the library.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot, 'extern', 'examples', 'shrlib'))
    loadlibrary('shrlibsample')
end
```

Create a structure.

```
S.p1 = 476;
S.p2 = -299;
S.p3 = 1000;
```

Call `addStructByRef`.

```
res = calllib('shrlibsample', 'addStructByRef', S)
res = 1177
```

MATLAB does not modify the contents of structure `S`, since it is not a pointer.

`S`

```
S = struct with fields:
  p1: 476
  p2: -299
  p3: 1000
```

Pass Pointer to Structure

This example shows how calling the `addStructByRef` function with a pointer modifies the fields in the input argument.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot, 'extern', 'examples', 'shrlib'))
```



```
    loadlibrary('shrlibsample')
end
```

Create a structure of type `c_struct`.

```
S.p1 = 20;
S.p2 = 99;
S.p3 = 3;
```

Create a pointer `sp` to the structure.

```
sp = libpointer('c_struct',S);
sp.Value
```

```
ans = struct with fields:
    p1: 20
    p2: 99
    p3: 3
```

Pass the pointer to the function.

```
res = calllib('shrlibsample','addStructByRef',sp)
res = 122
```

When you pass a pointer, the function modifies the fields in the structure it points to.

```
sp.Value
```

```
ans = struct with fields:
    p1: 5.5000
    p2: 1234
    p3: 12345678
```

See Also

`libpointer` | `libstruct`

Related Examples

- “Shared Library `shrlibsample`” on page 6-22

More About

- “Strategies for Passing Structures” on page 6-48
- “Limitations Using Structures” on page 6-11

Pass Enumerated Types Examples

In this section...

“readEnum Function” on page 6-31

“Display Enumeration Values” on page 6-32

readEnum Function

The readEnum function in the shrlibsample library displays a string that matches the input argument.

```
EXPORTED_FUNCTION char* readEnum(TEnum1 val)
{
    static char outputs[][20] = {
        {"You chose en1"},
        {"You chose en2"},
        {"You chose en4"},
        {"enum not defined"},
        {"ERROR"} };

    switch (val) {
        case en1: return outputs[0];
        case en2: return outputs[1];
        case en4: return outputs[2];
        default : return outputs[3];
    }
    return outputs[4];
}
```

The function signature is:

Return Type	Name	Arguments
cstring	readEnum	(Enum1)

The values for the Enum1 input are defined in the shrlibsample.h header file.

```
typedef enum Enum1 {en1 = 1, en2, en4 = 4} TEnum1;
```

Display Enumeration Values

This example shows how to pass enumeration values to the `readEnum` function in the `shrlibsample` library. Load the library.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

In MATLAB, you can express an enumerated type as either the enumeration string or its equivalent numeric value. Call `readEnum` with a string argument.

```
calllib('shrlibsample','readEnum','en4')
```

```
ans =
'You chose en4'
```

Call `readEnum` with the equivalent numeric argument. The `Enum1` definition declares enumeration `en4` equal to 4.

```
calllib('shrlibsample','readEnum',4)
```

```
ans =
'You chose en4'
```

See Also

Related Examples

- “Shared Library `shrlibsample`” on page 6-22

Pass Pointers Examples

In this section...

“multDoubleRef Function” on page 6-33
 “Pass Pointer of Type double” on page 6-33
 “Create Pointer Offset from Existing lib.pointer Object” on page 6-34
 “Multilevel Pointers” on page 6-35
 “allocateStruct and deallocateStruct Functions” on page 6-35
 “Pass Multilevel Pointer” on page 6-36
 “Return Array of Strings” on page 6-36

multDoubleRef Function

The multDoubleRef function in the shrlibsample library multiplies the input by 5.

```

EXPORTED_FUNCTION double *multDoubleRef(double *x)
{
    *x *= 5;
    return x;
}
  
```

The input is a pointer to a double, and the function returns a pointer to a double. The MATLAB function signature is:

Return Type	Name	Arguments
[lib.pointer, doublePtr]	multDoubleRef	(doublePtr)

Pass Pointer of Type double

This example shows how to construct and pass a pointer to C function multDoubleRef.

Load the library containing the function.

```

if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
  
```

Construct a pointer, `Xptr`, to the input argument, `X`.

```
X = 13.3;  
Xptr = libpointer('doublePtr',X);
```

Verify the contents of `Xptr`.

```
get(Xptr)  
  
Value: 13.3000  
DataType: 'doublePtr'
```

Call the function and check the results.

```
calllib('shrlibsample','multDoubleRef',Xptr);  
Xptr.Value  
  
ans = 66.5000
```

`Xptr` is a handle object. Copies of this handle refer to the same underlying object and any operations you perform on a handle object affect all copies of that object. However, `Xptr` is not a C language pointer. Although it points to `X`, it does not contain the address of `X`. The function modifies the `Value` property of `Xptr` but does not modify the value in the underlying object `X`. The original value of `X` is unchanged.

`X`

```
X = 13.3000
```

Create Pointer Offset from Existing `lib.pointer` Object

This example shows how to create a pointer to a subset of a MATLAB vector `X`. The new pointer is valid only as long as the original pointer exists.

Create a pointer to a vector.

```
X = 1:10;  
xp = libpointer('doublePtr',X);  
xp.Value  
  
ans = 1×10
```

```
1 2 3 4 5 6 7 8 9 10
```

Use the `lib.pointer` plus operator (+) to create a pointer to the last six elements of X.

```
xp2 = xp + 4;
xp2.Value
```

```
ans = 1x6
```

```
    5    6    7    8    9   10
```

Multilevel Pointers

Multilevel pointers are arguments that have more than one level of referencing. A multilevel pointer type in MATLAB uses the suffix `PtrPtr`. For example, use `doublePtrPtr` for the C argument `double **`.

When calling a function that takes a multilevel pointer argument, use a `lib.pointer` object and let MATLAB convert it to the multilevel pointer.

allocateStruct and deallocateStruct Functions

The `allocateStruct` function in the `shrlibsample` library takes a `c_structPtrPtr` argument.

```
EXPORTED_FUNCTION void allocateStruct(struct c_struct **val)
{
    *val=(struct c_struct*) malloc(sizeof(struct c_struct));
    (*val)->p1 = 12.4;
    (*val)->p2 = 222;
    (*val)->p3 = 333333;
}
```

The MATLAB function signatures are:

Return Type	Name	Arguments
<code>c_structPtrPtr</code>	<code>allocateStruct</code>	<code>(c_structPtrPtr)</code>
<code>voidPtr</code>	<code>deallocateStruct</code>	<code>(voidPtr)</code>

Pass Multilevel Pointer

This example shows how to pass a multilevel pointer to a C function.

Load the library containing `allocateStruct` and `deallocateStruct`.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Create a `c_structPtr` pointer.

```
sp = libpointer('c_structPtr');
```

Call `allocateStruct` to allocate memory for the structure.

```
res = calllib('shrlibsample','allocateStruct',sp)
```

```
res = struct with fields:
    p1: 12.4000
    p2: 222
    p3: 333333
```

Free the memory created by the `allocateStruct` function.

```
calllib('shrlibsample','deallocateStruct',sp)
```

Return Array of Strings

Suppose that you have a library, `myLib`, with a function, `acquireString`, that reads an array of strings. The function signature is:

Return Type	Name	Arguments
<code>char**</code>	<code>acquireString</code>	<code>(void)</code>

```
char** acquireString(void)
```

The following **pseudo-code** shows how to manipulate the return value, an array of pointers to strings.

```
ptr = calllib(myLib,'acquireString')
```


MATLAB creates a `lib.pointer` object `ptr` of type `stringPtrPtr`. This object points to the first string. To view other strings, increment the pointer. For example, to display the first three strings, type:

```
for index = 0:2
    tempPtr = ptr + index;
    tempPtr.Value
end

ans =
    'str1'
ans =
    'str2'
ans =
    'str3'
```

See Also

`libpointer`

Pass Arrays Examples

In this section...

“print2darray Function” on page 6-38

“Convert MATLAB Array to C-Style Dimensions” on page 6-38

“multDoubleArray Function” on page 6-39

“Preserve 3-D MATLAB Array” on page 6-40

print2darray Function

The `print2darray` function in the `shrlibsample` library displays the values of a 2-D array with three columns and a variable number of rows. The `my2d` parameter is a two-dimensional array of double. The `len` parameter is the number of rows.

```
EXPORTED_FUNCTION void print2darray(double my2d[][3],int len)
{
    int indxi,indxj;
    for(indxi=0;indxi<len;++indxi)
    {
        for(indxj=0;indxj<3;++indxj)
        {
            mexPrintf("%10g",my2d[indxi][indxj]);
        }
        mexPrintf("\n");
    }
}
```

Convert MATLAB Array to C-Style Dimensions

This example shows how to pass data stored columnwise in a MATLAB array to a C function that assumes a row-by-column format.

Load the library containing the `print2darray` function.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Create a MATLAB array with 4 rows and 3 columns.

```
m = reshape(1:12,4,3)
```

```
m = 4×3
```

```

1     5     9
2     6    10
3     7    11
4     8    12
```

Display the values. The first column is [1 4 7 10] instead of [1 2 3 4].

```
calllib('shrlibsample','print2darray',m,4)
```

```

1         2         3
4         5         6
7         8         9
10        11        12
```

Transpose m to get the desired result.

```
calllib('shrlibsample','print2darray',m',4)
```

```

1         5         9
2         6        10
3         7        11
4         8        12
```

multDoubleArray Function

The `multDoubleArray` function in the `shrlibsample` library multiplies each element of an array by three. The function uses a single subscript (linear indexing) to navigate the input array.

```

EXPORTED_FUNCTION void multDoubleArray(double *x,int size)
{
    /* Multiple each element of the array by 3 */
    int i;
    for (i=0;i<size;i++)
        *x++ *= 3;
}

```

The MATLAB function signature is:

Return Type	Name	Arguments
doublePtr	multDoubleArray	(doublePtr, int32)

Preserve 3-D MATLAB Array

This example shows how a C function changes the dimensions of a MATLAB array, and how to restore its shape.

Load the library.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Create a 2-by-5-by-2 input array and display its dimensions.

```
vin = reshape(1:20,2,5,2);
vs = size(vin)
```

```
vs = 1x3
     2     5     2
```

Call `multDoubleArray` to multiply each element. Display the dimensions of the output.

```
vout = calllib('shrlibsample','multDoubleArray',vin,20);
size(vout)
```

```
ans = 1x2
     2    10
```

Restore the original shape.

```
vout = reshape(vout,vs);
size(vout)
```

```
ans = 1x3
```

2 5 2

Iterate Through an Array

In this section...

“Create Cell Array from lib.pointer Object” on page 6-42

“Perform Pointer Arithmetic on Structure Array” on page 6-43

Create Cell Array from lib.pointer Object

This example shows how to create a MATLAB® cell array of character vectors, `mLStringArray`, from the output of the `getListOfStrings` function.

Load the `shrlibsample` library.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Call the `getListOfStrings` function to create an array of character vectors. The function returns a pointer to the array.

```
ptr = calllib('shrlibsample','getListOfStrings');
class(ptr)
```

```
ans =
'lib.pointer'
```

Create indexing variables to iterate through the arrays. Use `ptrindex` for the array returned by the function and `index` for the MATLAB array.

```
ptrindex = ptr;
index = 1;
```

Create the cell array of character vectors `mLStringArray`. Copy the output of `getListOfStrings` to the cell array.

```
% read until end of list (NULL)
while ischar(ptrindex.value{1})
    mLStringArray{index} = ptrindex.value{1};
    % increment pointer
    ptrindex = ptrindex + 1;
    % increment array index
```

```

        index = index + 1;
end

```

View the contents of the cell array.

```
mlStringArray
```

```
mlStringArray = 1x4 cell array
    {'String 1'}    {'String Two'}    {0x0 char}    {'Last string'}
```

Perform Pointer Arithmetic on Structure Array

This example shows how to use pointer arithmetic to access elements of a structure. The example creates a MATLAB structure, based on the `c_struct` definition in the `shrlibsample.h` header file.

Load the definition.

```

if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end

```

Create the MATLAB structure.

```
s = struct('p1',{1,2,3},'p2',{1.1,2.2,3.3},'p3',{0});
```

Create a pointer to the structure.

```
sptr = libpointer('c_struct',s);
```

Read the values of the first element.

```

v1 = sptr.Value
v1 = struct with fields:
    p1: 1
    p2: 1
    p3: 0

```

Read the values of the next element by incrementing the pointer.

```
sptr = sptr + 1;  
v2 = sptr.Value  
  
v2 = struct with fields:  
  p1: 2  
  p2: 2  
  p3: 0
```


Pointer Arguments

In this section...

“Pointer Arguments in C Functions” on page 6-45

“Put String into Void Pointer” on page 6-45

“Memory Allocation for External Library” on page 6-46

Pointer Arguments in C Functions

Many functions in external libraries pass arguments by reference. When you pass by reference, you pass a pointer to the value. In the function signature, pointer arguments have names ending in `Ptr` and `PtrPtr`. Although MATLAB does not support passing by reference, you can create a MATLAB argument, called a `lib.pointer` object, that is compatible with a C pointer. This object is an instance of the MATLAB `lib.pointer` class.

Often, you can simply pass a MATLAB variable (passing an argument by value), even when the signature for that function declares the argument to be a pointer. There are times, however, when it is useful to pass a `lib.pointer`.

- You want to modify the data in the input arguments.
- You are passing large amounts of data, and you want to control when MATLAB makes copies of the data.
- The library stores and uses the pointer so you want the MATLAB function to control the lifetime of the `lib.pointer` object.

Put String into Void Pointer

C represents characters as 8-bit integers. To use a MATLAB character array as an input argument, convert the string to the proper type and create a `voidPtr`. For example:

```
str = 'string variable';
vp = libpointer('voidPtr',[int8(str) 0]);
```

The syntax `[int8(str) 0]` creates the null-terminated string required by the C function. To read the string, and verify the pointer type, enter:

```
char(vp.Value)
vp.DataType
```

```
ans =  
string variable  
ans =  
voidPtr
```

MATLAB automatically converts an argument passed by value into an argument passed by reference when the external function prototype defines the argument as a pointer. Call a function that takes a `voidPtr` to a string as an input argument using the following syntax.

```
func_name([int8(str) 0])
```

Although MATLAB converts the argument from a value to a pointer, it must be of the correct type.

Memory Allocation for External Library

In general, MATLAB passes a valid memory address each time you pass a variable to a library function. Use a `lib.pointer` object in cases where the library stores the pointer and accesses the buffer over time. In these cases, ensure that MATLAB has control over the lifetime of the buffer and prevent copies of the data from being made. The following **pseudo-code** is an example of asynchronous data acquisition that shows how to use a `lib.pointer` in this situation.

Suppose an external library `myLib` has the following functions:

```
AcquireData(int points,short *buffer)  
IsAcquisitionDone(void)
```

where `buffer` is declared as follows:

```
short buffer[99]
```

First, create a `lib.pointer` to an array of 99 points:

```
BufferSize = 99;  
pBuffer = libpointer('int16Ptr',zeros(BufferSize,1));
```

Then, begin acquiring data and wait in a loop until it is done:

```
calllib('myLib','AcquireData',BufferSize,pbuffer)  
while (~calllib('myLib','IsAcquisitionDone'))  
    pause(0.1)  
end
```

The following statement reads the data in the buffer:

```
result = pBuffer.Value;
```

When the library is done with the buffer, clear the MATLAB variable:

```
clear pBuffer
```

See Also

`lib.pointer`

Structure Arguments

Structure Argument Requirements

When you pass a MATLAB structure to an external library function:

- Every MATLAB field name must match a field name in the library structure definition. Field names are case-sensitive.
- MATLAB structures cannot contain fields that are not in the library structure definition.
- If a MATLAB structure contains fewer fields than defined in the library structure, MATLAB sets undefined fields to zero.

You do not need to match the data types of numeric fields. The `calllib` function converts to the correct numeric type.

Find Structure Field Names

To determine the name and data type of structure fields, you can:

- Consult the library documentation.
- Look at the structure definition in the library header file.
- Use the `libstruct` function.

Strategies for Passing Structures

MATLAB automatically converts a structure to the library definition for that structure type. For most cases, such as working with small structures, this works fine.

However, when working with repeated calls that pass large structures, convert the structure manually before making any calls to external functions. You save processing time by converting the structure data only once at the start rather than at each function call. You can also save memory if the fields of the converted structure take up less space than the original MATLAB structure.

To convert manually, call the `libstruct` function to create a `libstruct` object. Although it is an object, it behaves like a MATLAB structure. The fields of the object are derived from an externally specified structure type.

See Also

libstruct

Related Examples

- “Add Values of Fields in Structure” on page 6-26
- “Preconvert MATLAB Structure Before Adding Values” on page 6-27

More About

- “Limitations Using Structures” on page 6-11

Explore libstruct Objects

This example shows how to display information about and modify a libstruct object, `c_struct`.

Load the `shrlibsample` library containing the `c_struct` definition.

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot, 'extern', 'examples', 'shrlib'))
    loadlibrary('shrlibsample')
end
```

Create the libstruct object. Object `sc` is an instance of a MATLAB class called `lib.c_struct`.

```
sc = libstruct('c_struct')
```

```
sc =  
  
    lib.c_struct
```

Set structure field values.

```
set(sc, 'p1', 100, 'p2', 150, 'p3', 200)
```

Display field values.

```
get(sc)  
  
    p1: 100  
    p2: 150  
    p3: 200
```

Modify values using MATLAB field structure syntax.

```
sc.p1 = 23;  
get(sc)  
  
    p1: 23  
    p2: 150  
    p3: 200
```

MATLAB Prototype Files

In this section...

“When to Use Prototype Files” on page 6-51

“How to Create Prototype Files” on page 6-51

“How to Specify Thunk Files” on page 6-52

“Deploy Applications That Use `loadlibrary`” on page 6-52

“`loadlibrary` in Parallel Computing Environment” on page 6-52

“Change Function Signature” on page 6-52

“Rename Library Function” on page 6-52

“Load Subset of Functions in Library” on page 6-52

“Call Function with Variable Number of Arguments” on page 6-53

When to Use Prototype Files

MATLAB provides a way to modify header file information by creating a prototype file, a file of MATLAB commands.

Like a header file, the prototype file contains the function signatures for the library. Here are some reasons for using a prototype file.

- To deploy applications that use `loadlibrary` (using MATLAB Compiler™).
- To use `loadlibrary` in a parallel computing environment (using Parallel Computing Toolbox™).
- To change signatures of the library functions.
- To rename some of the library functions.
- To use only a small percentage of the functions in the library you are loading.
- To use functions with a variable number of arguments.

You can change the prototypes by editing the prototype file and reloading the library.

How to Create Prototype Files

To create a prototype file, use the `mfilename` option of the `loadlibrary` function.

How to Specify Thunk Files

For information about default thunk file names, see `loadlibrary`. To change the name, use the `thunkfilename` option.

Deploy Applications That Use `loadlibrary`

To deploy a MATLAB application that uses `loadlibrary`, using MATLAB Compiler:

- Create a prototype file.
- For 64-bit applications, specify a thunk file.
- Include all the relevant files when creating the project with `mcc`.

`loadlibrary` in Parallel Computing Environment

To use `loadlibrary` in a parallel computing environment (using Parallel Computing Toolbox):

- Create a prototype file.
- For 64-bit applications, specify a thunk file.
- Make sure that all relevant files are accessible to all workers.

Change Function Signature

Edit the prototype file, changing the `fcns.LHS` or `fcns.RHS` field for that function. This edit changes the types of arguments on the left-hand side or right-hand side, respectively.

Rename Library Function

Edit the prototype file, defining the `fcns.alias` field for that function.

Load Subset of Functions in Library

Edit the prototype file, commenting out the unused functions. This edit reduces the amount of memory required for the library.

Call Function with Variable Number of Arguments

Create an alias function in a prototype file for each set of arguments you use to call the function.

Intro to MEX-Files

Choosing MEX Applications

You can call your own C, C++, or Fortran programs from the MATLAB command line as if they were built-in functions. These programs are called MEX functions. MEX functions are not appropriate for all applications. MATLAB is a high-productivity environment whose specialty is eliminating time-consuming, low-level programming in compiled languages like C or C++. In general, do your programming in MATLAB. Do not use MEX files unless your application requires it.

To create a MEX function, write your program using MATLAB APIs, then build it using the `mex` command. The APIs provide these features:

- Call MATLAB functions from the MEX function.
- Integrate seamlessly into MATLAB, getting inputs from and returning results to MATLAB.
- Support MATLAB data types.

C++ MEX Functions

As of MATLAB R2018a, write your C++ MEX functions using these APIs, which support C++11 programming features. These APIs, based on the `matlab::data::Array` class, provide better type safety, array bounds checking, and support for modern C++ constructs to simplify coding.

- “C++ MEX API” on page 9-14
- “MATLAB Data API”

For more information, see “C++ MEX Applications”.

C/C++ MEX Functions for MATLAB R2017b and Earlier

If your MEX functions must run in MATLAB R2017b or earlier, or if you prefer to work in the C language, then write your source files using functions in these libraries based on the `mxArray` data structure.

- C MEX API on “C MEX File Applications”
- “C Matrix API”

For more information, see “C MEX File Applications”.

Caution Do not mix functions in the C Matrix API with functions in the MATLAB Data API.

Fortran MEX Functions

To write Fortran MEX functions, use these APIs based on the `mxArray` data structure.

- “Fortran MEX API”
- “Fortran Matrix API”

For more information, see “Fortran MEX File Applications”.

MEX Terms

MEX stands for MATLAB executable and has different meanings, as shown in this table.

MEX Term	Definition
source MEX file	C, C++, or Fortran source code file.
binary MEX file or MEX function	Dynamically linked subroutine executed in the MATLAB environment.
MEX API	Functions in the C MEX API and Fortran MEX API to perform operations in the MATLAB environment.
mex build script	MATLAB function to create a binary file from a source file.

See Also

`matlab::data::Array` | `mex` | `mxArray`

More About

- “Tables of MEX Function Source Code Examples” on page 8-20

MEX File Placement

Put your MEX files in a folder on the MATLAB path. Alternatively, run MATLAB from the folder containing the MEX file. MATLAB runs functions in the current working folder before functions on the path.

To see the current folders on your path, use the `path` function. You can add new folders to the path either by using the `addpath` function, or by selecting **File > SetPath** to edit the path.

MEX Files on Windows Network Drives

Windows network drive file servers do not always report folder and file changes correctly. If you change a MEX file on a network drive and MATLAB does not use the latest changes, change folders away from and then back to the folder containing the file.

See Also

`addpath` | `path`

Use Help Files with MEX Functions

You can document the behavior of your MEX functions by writing a MATLAB script containing comment lines. The `help` command searches for a MATLAB script and displays the appropriate text.

For example, copy the following text from the `arrayProduct.c` MEX source file into a file, `arrayproduct.m`.

```
% arrayproduct.m Help file for arrayProduct MEX-file.  
% arrayProduct.c - example in MATLAB External Interfaces  
%  
% Multiplies an input scalar (multiplier)  
% times a 1xN matrix (inMatrix)  
% and outputs a 1xN matrix (outMatrix)  
%  
% The calling syntax is:  
%  
%         outMatrix = arrayProduct(multiplier, inMatrix)  
%  
%  
% This is a MEX-file for MATLAB.  
% Copyright 2007-2014 The MathWorks, Inc.  
%
```

When you type:

```
help arrayproduct
```

MATLAB displays the comments.

See Also

`help`

Related Examples

- “Document Build Information in the MEX File” on page 7-75
- “Add Help for Your Program”

MATLAB Data

The MATLAB Array

The MATLAB language works with a single object type: the MATLAB array. All MATLAB variables (including scalars, vectors, matrices, character arrays, cell arrays, structures, and objects) are stored as MATLAB arrays. In C/C++, the MATLAB array is declared to be of type `mxArray`. The `mxArray` structure contains the following information about the array:

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

To access the `mxArray` structure, use functions in the C or Fortran Matrix APIs. These functions allow you to create, read, and query information about the MATLAB data in your MEX files. Matrix APIs use the `mwSize` and `mwIndex` types to avoid portability issues and allow MEX source files to be compiled correctly on all systems.

Lifecycle of `mxArray`

Like MATLAB functions, a MEX-file gateway routine on page 8-2 passes MATLAB variables by reference. However, these arguments are C pointers. A pointer to a variable is the address (location in memory) of the variable. MATLAB functions handle data storage for you automatically. When passing data to a MEX-file, you use pointers, which follow specific rules for accessing and manipulating variables. For information about working with pointers, refer to a programming reference, such as *The C Programming Language* by Kernighan, B. W., and D. M. Ritchie.

Note Since variables use memory, you need to understand how your MEX-file creates an `mxArray` and your responsibility for releasing (freeing) the memory. This is important to prevent memory leaks. The lifecycle of an `mxArray`—and the rules for managing memory—depends on whether it is an input argument, output argument, or local variable. The

function you call to deallocate an `mxArray` depends on the function you used to create it. For more information, look up the functions to create arrays in the C Matrix API.

Input Argument `prhs`

An `mxArray` passed to a MEX-file through the `prhs` input parameter exists outside the scope of the MEX-file. Do not free memory for any `mxArray` in the `prhs` parameter. Also, `prhs` variables are read-only; do not modify them in your MEX-file.

Output Argument `plhs`

If you create an `mxArray` (allocate memory and create data) for an output argument, the memory and data exist beyond the scope of the MEX-file. Do not free memory on an `mxArray` returned in the `plhs` output parameter.

Local Variable

You allocate memory whenever you use an `mxCreate*` function to create an `mxArray` or when you call the `mxMalloc` and associated functions. After observing the rules for handling input and output arguments, the MEX-file should destroy temporary arrays and free dynamically allocated memory. To deallocate memory, use either `mxDestroyArray` or `mxFree`. For information about which function to use, see MX Matrix Library.

Data Storage

MATLAB stores data in a column-major (column-wise) numbering scheme, which is how Fortran stores matrices. MATLAB uses this convention because it was originally written in Fortran. MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on, through the last column.

For example, given the matrix:

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

```
a =  
    house  
    floor  
    porch
```

its dimensions are:

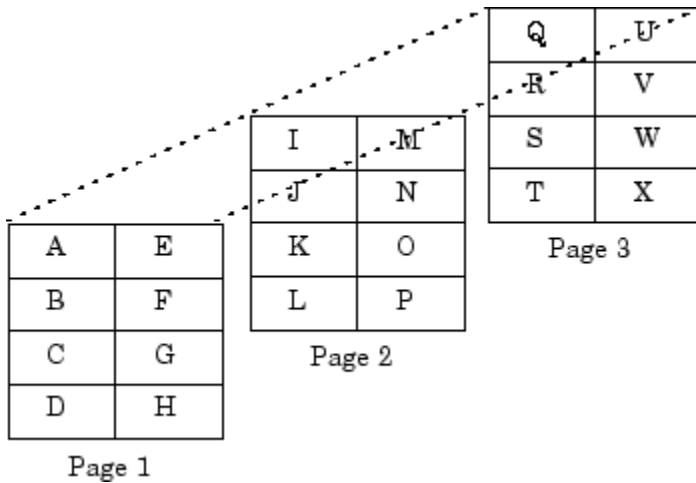
```
size(a)
```

```
ans =
     3     5
```

and its data is stored as:

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

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



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

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

The `mxCalcSingleSubscript` function creates the offset from the first element of an array to the desired element, using N-dimensional subscripting.

MATLAB Data Types

Complex Double-Precision Matrices

The most common data type in MATLAB is the complex double-precision, nonsparse matrix. These matrices are of type `double` and have dimensions *m*-by-*n*, where *m* is the number of rows and *n* is the number of columns. The data is stored as a vector of interleaved, double-precision numbers where the real and imaginary parts are stored next to each other. The pointer to this data is referred to as `pa` (pointer to array). To test for a noncomplex matrix, call `mxIsComplex`.

Before MATLAB Version 9.4 (R2018a), MATLAB used a *separate* storage representation. The data is stored as two vectors of double-precision numbers—one contains the real data and one contains the imaginary data. The pointers to this data are referred to as `pr` (pointer to real data) and `pi` (pointer to imaginary data), respectively. A noncomplex matrix is one whose `pi` is `NULL`. However, to test for a noncomplex matrix, call `mxIsComplex`.

Other Numeric Matrices

MATLAB supports single-precision floating-point and 8-, 16-, 32-, and 64-bit integers, both signed and unsigned.

Logical Matrices

The logical data type represents a logical `true` or `false` state using the numbers `1` and `0`, respectively. Certain MATLAB functions and operators return logical `1` or logical `0` to indicate whether a certain condition was found to be true or not. For example, the statement `(5 * 10) > 40` returns a logical `1` value.

MATLAB char Arrays

MATLAB char arrays store data as unsigned 16-bit integers. To convert a MATLAB char array to a C-style string, call `mxArrayToString`. To convert a C-style string to a char array, call `mxCreateString`.

Cell Arrays

Cell arrays are a collection of MATLAB arrays where each `mxArray` is referred to as a cell. Cell arrays allow MATLAB arrays of different types to be stored together. Cell arrays are stored in a similar manner to numeric matrices, except the data portion contains a

single vector of pointers to `mxArrays`. Members of this vector are called cells. Each cell can be of any supported data type, even another cell array.

Structures

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

Objects

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

Multidimensional Arrays

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

Empty Arrays

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

Sparse Matrices

Sparse matrices have a different storage convention from full matrices in MATLAB. The parameter `pa` is still an array of double-precision numbers or logical values, but this array contains only nonzero data elements.

There are three additional parameters: `nzmax`, `ir`, and `jc`. Use the `mwSize` and `mwIndex` types when declaring variables for these parameters.

- `nzmax` is an integer that contains the length of `ir` and `pa`. It is the maximum number of nonzero elements in the sparse matrix.
- `ir` points to an integer array of length `nzmax` containing the row indices of the corresponding elements in `pa`.

- `jc` points to an integer array of length $n+1$, where n is the number of columns in the sparse matrix. In C, the first element of an `mxAArray` has an index of 0. The `jc` array contains column index information. If the j th column of the sparse matrix has any nonzero elements, `jc[j]` is the index into `ir` and `pa` of the first nonzero element in the j th column. Index `jc[j+1] - 1` contains the last nonzero element in that column. For the j th column of the sparse matrix, `jc[j]` is the total number of nonzero elements in all preceding columns. The last element of the `jc` array, `jc[n]`, is equal to `nnz`, the number of nonzero elements in the entire sparse matrix. If `nnz` is less than `nzmax`, more nonzero entries can be inserted into the array without allocating more storage.

Using Data Types

You can write source MEX files, MAT-file applications, and engine applications in C/C++ that accept any class or data type supported by MATLAB (see “Data Types”). In Fortran, only the creation of double-precision n -by- m arrays and strings are supported. You use binary C/C++ and Fortran MEX files like MATLAB functions.

Caution MATLAB does not check the validity of MATLAB data structures created in C/C++ or Fortran using one of the Matrix Library create functions (for example, `mxCreatStructArray`). Using invalid syntax to create a MATLAB data structure can result in unexpected behavior in your C/C++ or Fortran program.

Declaring Data Structures

To handle MATLAB arrays, use type `mxAArray`. The following statement declares an `mxAArray` named `myData`:

```
mxAArray *myData;
```

To define the values of `myData`, use one of the `mxCreat*` functions. Some useful array creation routines are `mxCreatNumericArray`, `mxCreatCellArray`, and `mxCreatCharArray`. For example, the following statement allocates an m -by-1 floating-point `mxAArray` initialized to 0:

```
myData = mxCreatDoubleMatrix(m, 1, mxREAL);
```

C/C++ programmers should note that data in a MATLAB array is in column-major order. (For an illustration, see “Data Storage” on page 7-7.) Use the MATLAB `mxCreat*` array access routines to read data from an `mxAArray`.

Manipulating Data

The `mxGet*` array access routines get references to the data in an `mxArray`. Use these routines to modify data in your MEX file. Each function provides access to specific information in the `mxArray`. Some useful functions are `mxGetDoubles`, `mxGetComplexDoubles`, `mxGetM`, and `mxGetString`. Many of these functions have corresponding `mxSet*` routines to allow you to modify values in the array.

The following statements read the input `prhs[0]` into a C-style string `buf`.

```
char *buf;
int buflen;
int status;
buflen = mxGetN(prhs[0])*sizeof(mxChar)+1;
buf = mxMalloc(buflen);
status = mxGetString(prhs[0], buf, buflen);
```

explore Example

There is an example source MEX file included with MATLAB, called `explore.c`, that identifies the data type of an input variable. The source code for this example is in `matlabroot/extern/examples/mex`, where `matlabroot` represents the top-level folder where MATLAB is installed on your system.

Note In platform-independent discussions that refer to folder paths, this documentation uses the UNIX[®] convention. For example, a general reference to the `mex` folder is `matlabroot/extern/examples/mex`.

To build the example MEX file, first copy the file to a writable folder on your path.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'mex', 'explore.c'), '.', 'f')
```

Use the `mex` command to build the MEX file.

```
mex explore.c -R2018a
```

Type:

```
x = 2;
explore(x)
```

```
-----
Name: prhs[0]
```

```
Dimensions: 1x1  
Class Name: double
```

```
-----  
(1,1) = 2
```

explore accepts any data type. Try using explore with these examples:

```
explore([1 2 3 4 5])  
explore 1 2 3 4 5  
explore({1 2 3 4 5})  
explore(int8([1 2 3 4 5]))  
explore {1 2 3 4 5}  
explore(sparse(eye(5)))  
explore(struct('name', 'Joe Jones', 'ext', 7332))  
explore(1, 2, 3, 4, 5)  
explore(complex(3,4))
```

See Also

More About

- “Data Types”

Testing for Most-Derived Class

If you define functions that require inputs that are:

- MATLAB built-in types
- Not subclasses of MATLAB built-in types

use the following technique to exclude subclasses of built-in types from the input arguments.

- Define a cell array that contains the names of built-in types accepted by your function.
- Call `class` and `strcmp` to test for specific types in a MATLAB control statement.

The following code tests an input argument, `inputArg`:

```
if strcmp(class(inputArg), 'single')
    % Call function
else
    inputArg = single(inputArg);
end
```

Testing for a Category of Types

Suppose that you create a MEX function, `myMexFcn`, that requires two numeric inputs that must be of type `double` or `single`:

```
outArray = myMexFcn(a,b)
```

Define a cell array `floatTypes` that contains the words `double` and `single`:

```
floatTypes = {'double', 'single'};

% Test for proper types
if any(strcmp(class(a), floatTypes)) && ...
    any(strcmp(class(b), floatTypes))
    outArray = myMexFcn(a,b);
else
    % Try to convert inputs to avoid error
    ...
end
```


Another Test for Built-In Types

You can use `isobject` to separate built-in types from subclasses of built-in types. The `isobject` function returns `false` for instances of built-in types. For example:

```
% Create a int16 array  
a = int16([2,5,7,11]);  
isobject(a)
```

```
ans =  
    0
```

Determine if an array is one of the built-in integer types:

```
if isa(a, 'integer') && ~isobject(a)  
    % a is a built-in integer type  
    ...  
end
```

Build C MEX Function

This example shows how to build the example C MEX function `arrayProduct`. Use this example to verify the C build configuration for your system. For information about building C++ MEX functions, see “Build C++ MEX Programs” on page 9-10.

Verify that you have installed a Supported and Compatible Compilers. For information about installing a compiler, visit the vendor website.

```
mex -setup C
```

Copy the source MEX file to a writable folder on your path.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'mex', 'arrayProduct.c'), '.', 'f')
```

Call the `mex` command to build the function.

```
mex arrayProduct.c -R2018a
```

This command creates the file `arrayProduct.ext`, where `ext` is the value returned by the `mexext` function.

Test the function. The `arrayProduct` function takes a scalar and multiplies it by a 1xN matrix. Call `arrayProduct` like a MATLAB function.

```
s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A)

B =
    7.5000    10.0000    45.0000
```

See Also

`mex` | `mexext`

More About

- “Change Default Compiler” on page 7-18
- “Build C++ MEX Programs” on page 9-10

External Websites

- Supported and Compatible Compilers

Change Default Compiler

For information about changing the gcc/g++ compiler to a supported version on Linux platforms, see “Change Default gcc Compiler on Linux System” on page 7-20.

Windows Systems

MATLAB maintains separate default compiler options for C, C++, and Fortran language files. If you have multiple MATLAB-supported compilers for a language installed on your Windows system, then MATLAB selects one as the default compiler. To change the default, use the `mex -setup lang` command. MATLAB displays a message with links to select a different default compiler.

If you call `mex -setup` without the `lang` argument, then MATLAB displays information about the default C compiler. MATLAB also displays links to the other supported languages. To change the default for another language, select a link.

If you call `mex -setup` from an operating system prompt, MATLAB displays the same information. However, the messages do not contain links. Instead, MATLAB displays the appropriate `mex` command syntax for changing the default compiler. Copy the command and paste it into the operating system prompt.

The compiler you choose remains the default for that language until you call `mex -setup` to select a different default.

C Compilers

To change the default C compiler, at the MATLAB command prompt, type:

```
mex -setup
```

`mex -setup` defaults to information about the C compiler. Alternatively, type:

```
mex -setup c
```

C++ Compilers

To change the default C++ compiler, type:

```
mex -setup cpp
```

For an example, see “Choose a C++ Compiler” on page 8-27.

Fortran Compilers

To change the default Fortran compiler, type:

```
mex -setup Fortran
```

Use Non-Default Compiler on Mac Systems

If you have multiple versions of Xcode installed on your system, MATLAB uses the compiler defined by the `Xcode.app` application. You can use the compiler from an `Xcode.X.app`, where `Xcode.X.app` is the name you used to save a previously installed Xcode version.

Before starting MATLAB, from the Terminal type:

```
xcode-select -switch /Applications/Xcode.X.app/Contents/Developer
```

To see which Xcode MATLAB is using, at the Terminal type:

```
xcode-select -p
```

Do Not Use `mex -f` optionsfile Syntax

The `mex` command `-f` option to specify a build configuration file will be removed in a future release. Instead, use the workflows described in this topic for specifying a compiler.

See Also

`mex`

More About

- “Choose a C++ Compiler” on page 8-27
- “Change Default gcc Compiler on Linux System” on page 7-20

External Websites

- Supported and Compatible Compilers

Change Default gcc Compiler on Linux System

MATLAB supports only one compiler for each language on Linux platforms. If you have multiple compilers installed, the default compiler might not be the MATLAB-supported compiler. To determine the default gcc compiler for your system, in MATLAB, type:

```
!which gcc
```

To change the default compiler, choose one of these options.

- Change the system \$PATH variable. When you change the path, this compiler becomes the default for all applications on your system.

To change the \$PATH variable, add the folder containing the MATLAB-supported compiler to the beginning of the path. Refer to your operating system documentation for the correct command to use.

- Select a MATLAB-supported compiler when running the mex command. To change the compiler, use the varname variable set to GCC, in uppercase letters.

For example, if the currently supported gcc compiler is version 6.3 and it is installed in the /usr/bin/gcc-6.3 folder on your system, to build timestwo.c, type:

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','timestwo.c'),'.','f')  
mex -v GCC='/usr/bin/gcc-6.3' timestwo.c
```

Note The mex -setup command does not change the default compiler on Linux.

See Also

More About

- “Change Default Compiler” on page 7-18

External Websites

- Supported and Compatible Compilers
- How can I change my current GCC/G++ version to a supported one?

Custom Build with MEX Script Options

The `mex` build script is sufficient for building MEX files. Following are reasons that you might need more detailed information:

- You want to use an Integrated Development Environment (IDE), rather than the provided script, to build MEX files.
- You want to exercise more control over the build process than the script uses.

Use the `mex -v -n` options to display the build commands to configure an IDE. You can also use the `mex` script options to modify the build steps.

Include Files

Header files for the MATLAB API (MEX files, engine, and MAT-files). These files are in the `matlabroot\extern\include` folder.

- `matrix.h` — C/C++ header file containing a definition of the `mxArray` structure and function prototypes for matrix access routines.
- `mex.h` — Header file for building C/C++ MEX files. Contains function prototypes for `mex` routines.
- `engine.h` — C/C++ header file for MATLAB engine programs. Contains function prototypes for engine routines.
- `mat.h` — C/C++ header file for programs accessing MAT-files. Contains function prototypes for `mat` routines.
- `fintrf.h` — Header file for building Fortran MEX files. Contains function prototypes for `mex` routines.

See Also

`mex`

More About

- “Custom Linking to Required API Libraries” on page 1-6

Call LAPACK and BLAS Functions

You can call a LAPACK or BLAS function using a MEX file. To create a MEX file, you need C/C++ or Fortran programming experience and the software resources (compilers and linkers) to build an executable file. It also is helpful to understand how to use Fortran subroutines. MATLAB provides the `mwlapack` and `mwblas` libraries in `matlabroot/extern/lib`. To help you get started, there are source code examples in `matlabroot/extern/examples/refbook`.

To call LAPACK or BLAS functions:

- 1 Create a source MEX file containing the `mexFunction` gateway routine.
- 2 Make sure that you have a supported compiler for your platform. For an up-to-date list of supported compilers, see Supported and Compatible Compilers.
- 3 Build a binary MEX file using the `mex` command and the separate complex build flag `-R2017b`.
 - Link your source file to one or both of the libraries, `mwlapack` and `mwblas`.
 - The `mwlapack` and `mwblas` libraries only support 64-bit integers for matrix dimensions. Do not use the `-compatibleArrayDims` option.
 - To build a MEX file with functions that use complex numbers, see “Pass Separate Complex Numbers to Fortran Functions” on page 7-27.

Build `matrixMultiply` MEX Function Using BLAS Functions

This example shows how to build the example MEX file `matrixMultiply.c`, which uses functions from the BLAS library. To work with this file, copy it to a local folder. For example:

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'refbook', 'matrixMultiply.c'), '.')
```

The example files are read-only files. To modify an example, ensure that the file is writable by typing:

```
fileattrib('matrixMultiply.c', '+w')
```

To build the MEX file, type:

```
mex -v -R2017b matrixMultiply.c -lmwblas
```

To run the MEX file, type:


```
A = [1 3 5; 2 4 7];
B = [-5 8 11; 3 9 21; 4 0 8];
X = matrixMultiply(A,B)

X =
    24    35   114
    30    52   162
```

Preserve Input Values from Modification

Many LAPACK and BLAS functions modify the values of arguments passed to them. It is good practice to make a copy of arguments you can modify before passing them to these functions. For information about how MATLAB handles arguments to the `mexFunction`, see “Managing Input and Output Parameters” on page 8-3.

matrixDivide Example

This example calls the LAPACK function `dgesv` that modifies its input arguments. The code in this example makes copies of `prhs[0]` and `prhs[1]`, and passes the copies to `dgesv` to preserve the contents of the input arguments.

To see the example, open `matrixDivide.c` in the MATLAB Editor. To create the MEX file, copy the source file to a writable folder.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'refbook', 'matrixDivide.c'), '.')
```

To build the file, type:

```
mex -v -R2017b matrixDivide.c -lmwlapack
```

To test, type:

```
A = [1 2; 3 4];
B = [5; 6];
X = matrixDivide(A,B)

X =
   -4.0000
    4.5000
```

Pass Arguments to Fortran Functions from C/C++ Programs

The LAPACK and BLAS functions are written in Fortran. C/C++ and Fortran use different conventions for passing arguments to and from functions. Fortran functions pass

arguments by reference, while C/C++ functions pass arguments by value. When you pass by value, you pass a copy of the value. When you pass by reference, you pass a pointer to the value. A reference is also the address of the value.

When you call a Fortran subroutine, like a function from LAPACK or BLAS, from a C/C++ program, be sure to pass the arguments by reference. To pass by reference, precede the argument with an ampersand (&), unless that argument is already a reference. For example, when you create a matrix using the `mxGetDoubles` function, you create a reference to the matrix and do not need the ampersand before the argument.

In the following code snippet, variables `m`, `n`, `p`, `one`, and `zero` need the `&` character to make them a reference. Variables `A`, `B`, `C`, and `chn` are pointers, which are references.

```
/* pointers to input & output matrices*/
double *A, *B, *C;
/* matrix dimensions */
mwSignedIndex m,n,p;
/* other inputs to dgemm */
char *chn = "N";
double one = 1.0, zero = 0.0;

/* call BLAS function */
dgemm(chn, chn, &m, &n, &p, &one, A, &m, B, &p, &zero, C, &m);
```

matrixMultiply Example

The `matrixMultiply.c` example calls `dgemm`, passing all arguments by reference. To see the source code, open `matrixMultiply.c` in the MATLAB Editor. To build and run this example, see “Build `matrixMultiply` MEX Function Using BLAS Functions” on page 7-22.

Pass Arguments to Fortran Functions from Fortran Programs

You can call LAPACK and BLAS functions from Fortran MEX files. The following example takes two matrices and multiplies them by calling the BLAS routine `dgemm`. To run the example, copy the code into the editor and name the file `callldgemm.F`.

```
#include "fintrf.h"

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

```

mwPointer mxcreatedoublematrix
mwPointer mxgetpr
mwPointer A, B, C
mwSize mxgetm, mxgetn
mwSignedIndex m, n, p
mwSize numel
double precision one, zero, ar, br
character ch1, ch2

ch1 = 'N'
ch2 = 'N'
one = 1.0
zero = 0.0

A = mxgetpr(prhs(1))
B = mxgetpr(prhs(2))
m = mxgetm(prhs(1))
p = mxgetn(prhs(1))
n = mxgetn(prhs(2))

plhs(1) = mxcreatedoublematrix(m, n, 0.0)
C = mxgetpr(plhs(1))
numel = 1
call mxcopyprtoreal8(A, ar, numel)
call mxcopyprtoreal8(B, br, numel)

call dgemm(ch1, ch2, m, n, p, one, %val(A), m,
+         %val(B), p, zero, %val(C), m)

return
end

```

Link to the BLAS library, which contains the `dgemm` function.

```
mex -v -R2017b callldgemm.F -lmwblas
```

Modify Function Name on UNIX Systems

Add an underscore character following the function name when calling LAPACK or BLAS functions on a UNIX system. For example, to call `dgemm`, use:

```
dgemm_(arg1, arg2, ..., argn);
```

Or add these lines to your source code:

```
#if !defined(_WIN32)
#define dgemm dgemm_
#endif
```

See Also

Pass Separate Complex Numbers to Fortran Functions

MATLAB stores complex numbers the same way as Fortran, in one vector, `pa`, with the real and imaginary parts interleaved.

Before MATLAB Version 9.4 (R2018a), MATLAB stored complex numbers differently than Fortran, in separate, equal length vectors `pr` and `pi`. As a result, complex variables exchanged between those versions of MATLAB and a Fortran function are incompatible. MATLAB provides example conversion routines `mat2fort` and `fort2mat` that change the storage format of complex numbers to address this incompatibility. The `fort.h` header file defines the `mat2fort` and `fort2mat` functions. The source code is in the `fort.c` file.

- `mat2fort` — Convert MATLAB separate complex matrix to Fortran complex storage.
- `fort2mat` — Convert Fortran complex storage to MATLAB separate real and imaginary parts.

To use these routines:

- 1 Include the `fort.h` header file in your source file, using the statement `#include "fort.h"`.
- 2 Link the `fort.c` file with your program. Specify the full path, `matlabroot/extern/examples/refbook` for `fort.c` in the build command.
- 3 To indicate the header file, use the `-Ipathname` switch. Specify the full path, `matlabroot/extern/examples/refbook` for `fort.h` in the build command.
- 4 When you specify the full path, replace the term `matlabroot` with the actual folder name.
- 5 Build the function using the `mex -R2017b` option.

Handling Complex Number Input Values

It is unnecessary to copy arguments for functions that use complex number input values. The `mat2fort` conversion routine creates a copy of the arguments for you. For information, see “Preserve Input Values from Modification” on page 7-23.

Handling Complex Number Output Arguments

For complex variables returned by a Fortran function, do the following:

- 1 When allocating storage for the variable, allocate a real variable with twice as much space as you would for a variable of the same size. Do this because the returned variable uses the Fortran format, which takes twice the space. See the allocation of `zout` in the example.
- 2 To make the variable compatible with MATLAB, use the `fort2mat` function.

Pass Complex Variables — `matrixDivideComplex`

This example shows how to call a function, passing complex `prhs[0]` as input and receiving complex `plhs[0]` as output. Temporary variables `zin` and `zout` contain the input and output values in Fortran format. To see the example, open `matrixDivideComplex.c` in the MATLAB Editor. To create the MEX file, copy the source file to a writable folder.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'refbook', 'matrixDivideComplex.c'), '.');
```

Create variables locating the `fort.c` file and its header file.

```
fortfile = fullfile(matlabroot, 'extern', 'examples', 'refbook', 'fort.c');  
fortheaderdir = fullfile(matlabroot, 'extern', 'examples', 'refbook');
```

Build the MEX function.

```
mex('-v', '-R2017b', ['-I' fortheaderdir], 'matrixDivideComplex.c', fortfile, '-lmwlapack')
```

Test the function.

```
Areal = [1 2; 3 4];  
Aimag = [1 1; 0 0];  
Breal = [5; 6];  
Bimag = [0; 0];  
Acomplex = complex(Areal, Aimag);  
Bcomplex = complex(Breal, Bimag);  
X = matrixDivideComplex(Acomplex, Bcomplex)
```

```
X =  
-4.4000 + 0.8000i  
4.8000 - 0.6000i
```

Handle Fortran Complex Return Type — `dotProductComplex`

Some level 1 BLAS functions (for example, `zdotu` and `zdotc`) return a double complex type, which the C language does not support. The following C MEX file,

`dotProductComplex.c`, shows how to handle the Fortran complex return type for function `zdotu`. To see the example, open `dotProductComplex.c` in the MATLAB Editor.

The calling syntax for a C program calling a Fortran function that returns a value in an output argument is platform-dependent. On the Windows and Mac platforms, pass the return value as the first input argument. MATLAB provides a macro, `FORTRAN_COMPLEX_FUNCTIONS_RETURN_VOID`, to handle these differences.

The `dotProductComplex` example computes the dot product X of each element of two complex vectors A and B . The calling syntax is:

```
X = dotProductComplex(A,B)
```

where A and B are complex vectors of the same size and X is a complex scalar.

For example, to build:

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','dotProductComplex.c'),'.')
fortfile = fullfile(matlabroot,'extern','examples','refbook','fort.c');
fortheaderdir = fullfile(matlabroot,'extern','examples','refbook');
mex('-v','-R2017b',['-I' fortheaderdir],'dotProductComplex.c',fortfile,'-lmwblas')
```

To test, type;

```
a1 = [1+2i; 2+3i];
b1 = [-1+2i; -1+3i];
X = dotProductComplex(a1,b1)
```

```
X =
   -16.0000 + 3.0000i
```

Symmetric Indefinite Factorization Using LAPACK – `utdu_slv`

The example `utdu_slv.c` calls LAPACK functions `zhesvx` and `dsysvx`. To see the example, open `utdu_slv.c` in the MATLAB Editor. To create the MEX file, copy the source file to a writable folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','utdu_slv.c'),'.')
```

To build:

```
fortheaderdir = fullfile(matlabroot,'extern','examples','refbook');
mex('-v','-R2017b',['-I' fortheaderdir],'utdu_slv.c',fortfile,'-lmwlapack')
```

See Also

MATLAB Support for Interleaved Complex API in MEX Functions

When a matrix of complex numbers is represented in computer memory, there are several choices for the location and order of storage. Before MATLAB Version 9.4 (R2018a), MATLAB used a separate storage representation. The real and the imaginary parts of the numbers were stored separately. MATLAB uses an interleaved storage representation for C and Fortran language MEX files, where the real and imaginary parts are stored together. Interleaved complex refers to this representation.

Since many mathematical libraries use an interleaved complex representation, using the same representation in your MEX functions eliminates the need to translate data. This simplifies your code and potentially speeds up the processing when large data sets are involved.

This change does not affect the MATLAB language. You can continue to use the functionality described in “Complex Numbers” without any modification of your functions and scripts.

Separate Complex API and Interleaved Complex API

There are two versions of the C Matrix API and “Fortran Matrix API”.

- The separate complex API contains the C and Fortran Matrix API functionality in MATLAB R2017b and earlier.
- The interleaved complex API contains the C Matrix API functionality as of MATLAB R2018a and the Fortran Matrix API functionality as of MATLAB R2018a Update 3.

To build MEX files with the interleaved complex API, use the `mex` release-specific build option `-R2018a`. To build MEX files with the separate complex API, use the `-R2017b` build option. The `mex` command uses the separate complex API by default. However, in a future version of MATLAB, `mex` will use the interleaved complex API (`-R2018a` option) by default and then you need to modify your build command. Therefore, to ensure the desired behavior across versions of MATLAB, add the `-R2017b` option to your existing build scripts. To write code to support both APIs, see “Add `MX_HAS_INTERLEAVED_COMPLEX` to Support Both Complex Number Representations” on page 7-38.

Note To run a Fortran MEX file built with the interleaved complex API in MATLAB R2018a, you must use MATLAB R2018a Update 3.

Matrix API Changes Supporting Interleaved Complex

The following changes to the C and Fortran Matrix APIs support interleaved complex.

- New typed data access functions. For more information, see “Typed Data Access in C MEX Files” on page 8-65.

MATLAB mxArray Types	C Typed Data Access Functions	Fortran Typed Data Access Functions
mxDOUBLE_CLASS	mxGetDoubles mxSetDoubles mxGetComplexDoubles mxSetComplexDoubles	mxGetDoubles mxSetDoubles mxGetComplexDoubles mxSetComplexDoubles
mxSINGLE_CLASS	mxGetSingles mxSetSingles mxGetComplexSingles mxSetComplexSingles	mxGetSingles mxSetSingles mxGetComplexSingles mxSetComplexSingles
mxINT8_CLASS	mxGetInt8s mxSetInt8s mxGetComplexInt8s mxSetComplexInt8s	mxGetInt8s mxSetInt8s mxGetComplexInt8s mxSetComplexInt8s
mxUINT8_CLASS	mxGetUint8s mxSetUint8s mxGetComplexUint8s mxSetComplexUint8s	mxGetUint8s mxSetUint8s mxGetComplexUint8s mxSetComplexUint8s
mxINT16_CLASS	mxGetInt16s mxSetInt16s mxGetComplexInt16s mxSetComplexInt16s	mxGetInt16s mxSetInt16s mxGetComplexInt16s mxSetComplexInt16s
mxUINT16_CLASS	mxGetUint16s mxSetUint16s mxGetComplexUint16s mxSetComplexUint16s	mxGetUint16s mxSetUint16s mxGetComplexUint16s mxSetComplexUint16s

MATLAB mxArray Types	C Typed Data Access Functions	Fortran Typed Data Access Functions
mxINT32_CLASS	mxGetInt32s mxSetInt32s mxGetComplexInt32s mxSetComplexInt32s	mxGetInt32s mxSetInt32s mxGetComplexInt32s mxSetComplexInt32s
mxUINT32_CLASS	mxGetUint32s mxSetUint32s mxGetComplexUint32s mxSetComplexUint32s	mxGetUint32s mxSetUint32s mxGetComplexUint32s mxSetComplexUint32s
mxINT64_CLASS	mxGetInt64s mxSetInt64s mxGetComplexInt64s mxSetComplexInt64s	mxGetInt64s mxSetInt64s mxGetComplexInt64s mxSetComplexInt64s
mxUINT64_CLASS	mxGetUint64s mxSetUint64s mxGetComplexUint64s mxSetComplexUint64s	mxGetUint64s mxSetUint64s mxGetComplexUint64s mxSetComplexUint64s

- Matrix API functions added to the -R2018a API:
 - mxMakeArrayComplex (C) and mxMakeArrayComplex (Fortran)
 - mxMakeArrayReal (C) and mxMakeArrayReal (Fortran)
- Matrix API functions not supported in the -R2018a API:
 - mxGetPi (C) and mxGetPi (Fortran)
 - mxGetImagData (C) and mxGetImagData (Fortran)
 - mxSetPi (C) and mxSetPi (Fortran)
 - mxSetImagData (C) and mxSetImagData (Fortran)
- The behavior of these Matrix API functions changes in the -R2018a API. In addition, these functions are to be phased out.
 - mxGetPr (C) and mxGetPr (Fortran)
 - mxSetPr (C) and mxSetPr (Fortran)
- The behavior of these Matrix API functions changes in the -R2018a API:
 - mxGetData (C) and mxGetData (Fortran)

- `mxSetData` (C) and `mxSetData` (Fortran)
- `mxGetElementSize` (C) and `mxGetElementSize` (Fortran)
- The behavior of these Fortran Matrix API functions changes in the -R2018a API:
 - `mxCopyComplex16ToPtr` (Fortran)
 - `mxCopyPtrToComplex16` (Fortran)
 - `mxCopyComplex8ToPtr` (Fortran)
 - `mxCopyPtrToComplex8` (Fortran)

Writing MEX Functions with Interleaved Complex API

To learn how to write MEX functions using the interleaved complex API, see “Handle Complex Data in C MEX File” on page 8-44 in C applications or “Handle Complex Fortran Data” on page 10-31.

MEX Functions Created in MATLAB R2017b and Earlier

If you already build MEX functions, MEX S-functions, or standalone MATLAB engine and MAT-file applications, then you should review the “Do I Need to Upgrade My MEX Files to Use Interleaved Complex API?” on page 7-45 topic. To help transition your MEX files and S-functions to the interleaved complex API, MATLAB maintains a compatibility interface. To build MEX files with the compatibility interface, use the `mex -R2017b` flag. If your code processes complex numbers, you might notice a performance impact as MATLAB adapts your data to the interleaved complex format.

The `mex` command uses the -R2017b API by default. However, in a future version of MATLAB, `mex` will use the interleaved complex API -R2018a by default and then you need to modify your build command. Therefore, to ensure the desired behavior across versions of MATLAB, add the compatibility flag -R2017b to your build scripts.

See Also

More About

- “Upgrade MEX Files to Use Interleaved Complex API” on page 7-36
- C Matrix API

- “Fortran Matrix API”

Upgrade MEX Files to Use Interleaved Complex API

This topic describes how to upgrade your MEX files to use the interleaved complex API. You can continue to use the separate complex API by calling the `mex` command with the `-R2017b` option. However, for more information about using this option, see “Do I Need to Upgrade My MEX Files to Use Interleaved Complex API?” on page 7-45

Note If you build your MEX files using the `mex` command with the `-compatibleArrayDims` option, then you first must update the source code to use the 64-bit API. For information, see “Upgrade MEX Files to Use 64-Bit API” on page 7-51.

To update your MEX source code, use the following checklist.

- 1** Review your code for usage of `pr` and `pi` pointers, the pointers returned by the `mxGetPr/mxGetPi` and `mxGetData/mxGetImagData` functions. In the interleaved complex API, there is one pointer, `pa`, the value returned by `mxGetDoubles` and the other typed data functions. It is important to check an input array for complexity before attempting to read the data. Calls to `mxGetPr` and `mxGetData` on complex arrays return different results in the interleaved complex API than in the separate complex API.
- 2** Prepare your code before editing.

Before modifying your code, verify that the MEX function works with the `-R2017b` API. At a minimum, build a list of expected inputs and outputs, or create a full test suite. Use these tests to compare the results with the updated source code. The results should be identical.

Back up all source, binary, and test files.

- 3** Iteratively refactor your existing code by checking for the following conditions.
 - a** “Check Array Complexity Using `mxIsComplex`” on page 7-37
 - b** “Add `MX_HAS_INTERLEAVED_COMPLEX` to Support Both Complex Number Representations” on page 7-38
 - c** “Use Typed Data Access Functions” on page 7-39
 - d** “Handle Complex `mxArrays`” on page 7-39
 - e** “Replace Separate Complex Functions” on page 7-42

- f** “Calculate Array Data Size with `mxGetElementSize`” on page 7-42
- g** “Consider Replacing To-Be-Phased-Out Functions” on page 7-42
- 4** After each change, compile using the interleaved complex API. To build `myMexFile.c`, type:

```
mex -R2018a myMexFile.c
```

To build `myMexFile.F`, type:

```
mex -R2018a myMexFile.F
```

- 5** Resolve failures and warnings.
- 6** Test after each refactoring.

Compare the results of running your MEX function compiled with the interleaved complex API with the results from your original binary. If there are any differences or failures, use a debugger to investigate the cause. For information on the capabilities of your debugger, refer to your compiler documentation.

- 7** “Add Build Information to MEX Help File” on page 7-43

Check Array Complexity Using `mxIsComplex`

If your code calls the `mxGetPi` function to determine if an array has complex elements, use the `mxIsComplex` function instead. This function builds with both the `-R2017b` and `-R2018a` APIs. Search your code for the following patterns.

Replace C Source Code:	With:
<pre>mxArray *pa; ... if (mxGetPi(pa)) { /* process complex array */ }</pre>	<pre>mxArray *pa; ... if (mxIsComplex(pa)) { /* process complex array */ }</pre>
<pre>double *ptr; ptr = mxGetPi(pa); if (ptr != NULL) { /* process complex array */ }</pre>	

Add MX_HAS_INTERLEAVED_COMPLEX to Support Both Complex Number Representations

To write code that builds with both the -R2017b and -R2018a APIs, add the `MX_HAS_INTERLEAVED_COMPLEX` macro. This macro returns `true` if you build the MEX file with the -R2018a option.

Wrapping the following code in an `#if MX_HAS_INTERLEAVED_COMPLEX` statement ensures that this code will build with either the -R2017b or -R2018a mex option. However, in this example, there is no code to execute when built with -R2018a.

Replace C Source Code:	With:
<pre>static void analyze_double(const mxArray *array_ptr) { mwSize total_num_of_elements, index; double *pr, *pi; total_num_of_elements = mxGetNumberOfElements(array_ptr); pr = mxGetPr(array_ptr); pi = mxGetPi(array_ptr); for (index=0; index<total_num_of_elements; index++) { if (mxIsComplex(array_ptr)) { mexPrintf("%g + %gi\n", *pr++, *pi++); } else { mexPrintf("%g\n", *pr++); } } }</pre>	<pre>static void analyze_double(const mxArray *array_ptr) { mwSize total_num_of_elements, index; total_num_of_elements = mxGetNumberOfElements(array_ptr); #if MX_HAS_INTERLEAVED_COMPLEX /* add interleaved complex API code here */ #else /* separate complex API processing */ #endif double *pr, *pi; pr = mxGetPr(array_ptr); pi = mxGetPi(array_ptr); for (index=0; index<total_num_of_elements; index++) { if (mxIsComplex(array_ptr)) { mexPrintf("%g + %gi\n", *pr++, *pi++); } else { mexPrintf("%g\n", *pr++); } } #endif }</pre>

Replace Fortran Source Code:	With:
<pre>mwPointer prhs(*), pr pr = mxGetPr(prhs(1))</pre>	<pre>mwPointer prhs(*), pr #if MX_HAS_INTERLEAVED_COMPLEX pr = mxGetDoubles(prhs(1)) #else pr = mxGetPr(prhs(1)) #endif</pre>

Use Typed Data Access Functions

To use the `mxGetData` and `mxGetImagData` functions, you must verify the type of the input `mxArray` and manually cast the pointer output to the correct type. The typed data access functions verify the type of the array and return the correct pointer type. When you use the `mxGetInt16s` and `mxGetComplexInt16s` functions in the following code to process an `int16` array, you do not need to remember the corresponding C type, `short int`.

Replace C Source Code:	With:
<pre>static void analyze_int16(const mxArray *array_ptr) { short int *pr, *pi; pr = (short int *)mxGetData(array_ptr); pi = (short int *)mxGetImagData(array_ptr); if (mxIsComplex(array_ptr)) { /* process complex data *pr,*pi */ } else { /* process real data *pr */ } }</pre>	<pre>static void analyze_int16(const mxArray *array_ptr) { mxComplexInt16 *pc; mxInt16 *p; if (mxIsComplex(array_ptr)) { pc = mxGetComplexInt16s(array_ptr); /* process complex data (*pc).real,(*pc).imag */ } else { p = mxGetInt16s(array_ptr); /* process real data *p */ } }</pre>

Handle Complex mxArrays

The following examples show how MATLAB uses one array variable to represent a complex array.

Complex C mxArray

Suppose that you have the following complex mxArray variables and want to add the real numbers and the imaginary numbers of x and y to create array z. Arrays x and y are the same size.

```
mxArray * x, y, z;
```

Instead of creating two pointers xr and xi for array x, create one pointer xc of type mxComplexDouble. To access the real and imaginary parts of element xc[i], use xc[i].real and xc[i].imag.

Replace C Source Code:	With:
<pre>double *xr, *xi, *yr, *yi, *zr, *zi; /* get pointers to the real and imaginary parts of the arrays */ xr = mxGetPr(x); xi = mxGetPi(x); yr = mxGetPr(y); yi = mxGetPi(y); zr = mxGetPr(z); zi = mxGetPi(z); ... /* perform addition on element i */ zr[i] = xr[i] + yr[i]; zi[i] = xi[i] + yi[i];</pre>	<pre>/* get pointers to the complex arrays */ mxComplexDouble * xc = mxGetComplexDoubles(x); mxComplexDouble * yc = mxGetComplexDoubles(y); mxComplexDouble * zc = mxGetComplexDoubles(z); ... /* perform addition on element i */ zc[i].real = xc[i].real + yc[i].real; zc[i].imag = xc[i].imag + yc[i].imag;</pre>

The following code copies an mxArray into an output argument. The code shows how to test for and copy complex arrays.

Replace C Source Code:	With:
<pre>mxGetPr(plhs[0])[0] = mxGetPr(prhs[0])[index]; if (mxIsComplex(prhs[0])) { mxGetPi(plhs[0])[0] = mxGetPi(prhs[0])[index]; } }</pre>	<pre>if (mxIsComplex(prhs[0])) { mxGetComplexDoubles(plhs[0])[0] = mxGetComplexDoubles(prhs[0])[index]; } else { mxGetDoubles(plhs[0])[0] = mxGetDoubles(prhs[0])[index]; } }</pre>

Complex Fortran mxArray

Suppose that you have two complex double mxArray and want to pass them to a Fortran function with input arguments x and y defined as follows.

```
complex*16 x(*), y(*)
```

Instead of separately converting the real and imaginary parts of each mxArray, use the `mxGetComplexDoubles` function.

Replace Fortran Source Code:	With:
<pre> mwPointer mxGetPr, mxGetPi C Copy the data into native COMPLEX Fortran integers call mxCopyPtrToComplex16(+ mxGetPr(prhs(1)), + mxGetPi(prhs(1)),x,nx) call mxCopyPtrToComplex16(+ mxGetPr(prhs(2)), + mxGetPi(prhs(2)),y,ny) </pre>	<pre> mwPointer mxGetComplexDoubles integer*4 status C Copy the data into native COMPLEX Fortran status = + mxCopyPtrToComplex16(mxGetComplexDoubles C Test status for error conditions status = + mxCopyPtrToComplex16(mxGetComplexDoubles C Test status for error conditions </pre>

Maintain Complexity of mxArray

This C code snippet shows how to convert a real, double, input array `prhs[0]` into a complex array. The following code sets up the variables used to fill the complex part of the array with consecutive numbers.

```
// code to check number of arguments and expected types
mwSize rows = mxGetM(prhs[0]);
mwSize cols = mxGetN(prhs[0]);
mwSize sz = mxGetElementSize(prhs[0]);
```

The following code shows how to use `mxMakeArrayComplex` to convert a real, double, input array to an interleaved complex mxArray. For more examples, see `mxMakeArrayComplex (C)`.

Replace C Source Code:	With:
<pre> plhs[0] = mxDuplicateArray(prhs[0]); mxDouble *dc = (mxDouble*)mxMalloc(rows*cols*sz); mxSetImagData(plhs[0], dc); for (int i = 0 ; i < rows*cols ; i++) { dc[i] = i+1; } </pre>	<pre> plhs[0] = mxDuplicateArray(prhs[0]); if (mxMakeArrayComplex(plhs[0])) { mxComplexDouble *dt = mxGetComplexDoubles(plhs[0]); for (int i = 0 ; i < rows*cols ; i++) { dt[i].imag = i+1; } } </pre>

Replace Separate Complex Functions

The following functions are not in the interleaved complex API. You must replace them with interleaved complex functions when handling complex data.

- `mxGetPi`
- `mxSetPi`
- `mxGetImagData`
- `mxSetImagData`

You can replace calls to `mxGetPr` and `mxGetPi` with a call to `mxGetComplexDoubles`. This function verifies that your array contains elements of type `mxComplexDouble`. Likewise, `mxSetComplexDoubles` replaces `mxSetPr` and `mxSetPi`.

The `mxGetData` and `mxGetImagData` functions do not check the type of the array. Instead, you must cast the return value to the pointer type that matches the type specified by the input. Replace calls to `mxGetData` and `mxGetImagData` with the single, appropriate, typed data access function, for example, `mxGetComplexInt64s`.

Replace C Source Code:	With:
<pre>mxArray *pa; mwSize numElements; int64_T *pr, *pi; pr = (int64_T *)mxGetData(pa); pi = (int64_T *)mxGetImagData(pa); numElements = mxGetNumberOfElements(pa);</pre>	<pre>mxArray *pa; mwSize numElements; mxComplexInt64 *pc; pc = mxGetComplexInt64s(pa); numElements = mxGetNumberOfElements(pa);</pre>

Calculate Array Data Size with `mxGetElementSize`

In the -R2018a API, the `mxGetElementSize` (C) function returns `sizeof(std::complex<T>)` for a complex `mxArray` with data type `T`. This value is twice the value returned by the function in the -R2017b API. Similarly, `mxGetElementSize` (Fortran) in the -R2018a API returns twice the value as the function in the -R2017b API.

Consider Replacing To-Be-Phased-Out Functions

The following functions are in both the -R2017b and the -R2018a APIs. While you do not need replace them with typed data access functions, the typed data functions provide

type checking. Also, if you use `mxGetPr`, you might choose `mxGetPi` for any complex array processing. This code pattern causes errors when writing -R2018a MEX functions.

- `mxGetPr`
- `mxSetPr`
- `mxGetData` (C) and `mxGetData` (Fortran) - for numeric arrays
- `mxSetData` (C) and `mxSetData` (Fortran) - for numeric arrays

You can replace calls to `mxGetPr` with a call to `mxGetDoubles`. This function verifies that your array contains elements of type `mxDouble`. Likewise, `mxSetDoubles` replaces `mxSetPr`. To replace calls to `mxGetData` and `mxSetData` functions, choose the appropriate typed data access function, for example, `mxGetInt64s` and `mxSetInt64s`.

Replace C Source Code:	With:
<pre>double *y; /* create a pointer y to input matrix y = mxGetPr(prhs[1]);</pre>	<pre>mxDouble *p; p = mxGetDoubles(prhs[1]);</pre>

Replace Fortran Source Code:	With:
<pre>mwPointer pr mwPointer mxGetPr C Create a pointer to input matrix pr = mxGetPr(prhs(1))</pre>	<pre>mwPointer pr mwPointer mxGetDoubles pr = mxGetDoubles(prhs(1))</pre>

Add Build Information to MEX Help File

Consider creating a help file, described in “Use Help Files with MEX Functions” on page 7-5, that contains build information. For example, create a file `displayTypesafeNumeric.m` containing the following text.

```
% displayTypesafeNumeric.m Help file for displayTypesafeNumeric C MEX function
%
% Use the following command to build this MEX file:
% mex -R2018a displayTypesafeNumeric.c
```

At the MATLAB command prompt, type:

```
help displayTypesafeNumeric
```

```
displayTypesafeNumeric.m Help file for displayTypesafeNumeric C MEX function

Use the following command to build this MEX file:
mex -R2018a displayTypesafeNumeric.c
```

See Also

mex

More About

- “Troubleshooting MEX API Incompatibilities” on page 7-48
- “Do I Need to Upgrade My MEX Files to Use Interleaved Complex API?” on page 7-45

Do I Need to Upgrade My MEX Files to Use Interleaved Complex API?

You do not need to update your MEX source code to continue to build your MEX files. The `mex` command uses the `-R2017b` API by default. However, in a future version of MATLAB, `mex` will use the interleaved complex API `-R2018a` by default and then you need to modify your build command. Therefore, to ensure the desired behavior across versions of MATLAB, add the compatibility flag `-R2017b` to your build scripts.

Can I Run Existing MEX Functions?

You can run existing binary MEX files without upgrading the files for use with the interleaved complex API. However, other incompatibilities might prevent execution of an existing MEX function. If your function does not execute properly, search for `mex` in the relevant MATLAB release notes and review the Compatibility Considerations topics.

Must I Update My Source MEX Files?

State of Your Source Code	Next Action
My MEX functions do not use complex numbers.	<p>Check that your functions properly handle any complex data input. Calls to the <code>mxGetPr (C)/mxGetPr (Fortran)</code> and <code>mxGetData (C)/mxGetData (Fortran)</code> are not recommended for complex arrays.</p> <p>MathWorks recommends that you update your code to use the <code>MX_HAS_INTERLEAVED_COMPLEX</code> macro or build using the <code>mex -R2017b</code> option to ensure the desired behavior across versions of MATLAB.</p> <p>If you use <code>mxGetData</code> or <code>mxSetData</code>, consider replacing them with typed data access functions. For more information, see Using the interleaved complex API.</p>

State of Your Source Code	Next Action
I do not plan to update my code.	<p>If your MEX functions use complex numbers, then you have chosen to opt out. MathWorks recommends that you build using the compatibility flag <code>-R2017b</code>.</p> <p>If your code processes complex numbers, you might notice a performance impact as MATLAB accesses the compatibility interface.</p>
I want to update my code. Where do I start?	To update source code, see “Upgrade MEX Files to Use Interleaved Complex API” on page 7-36.
I use complex numbers in MEX functions, but do not have access to the source code.	Ask the owner of the source code to follow the steps in “Upgrade MEX Files to Use Interleaved Complex API” on page 7-36.
I use complex numbers with third-party libraries. My MEX code is responsible for transforming the MATLAB representation of complex numbers to the library's representation of complex numbers.	<p>Identify the library's representation of complex numbers. Sometimes the representation might be closer to the interleaved representation used in MATLAB.</p> <p>In other cases, libraries have options for representing complex numbers in memory. If this is so, refer to the vendor documentation and choose the representation that most closely matches the MATLAB interleaved representation.</p>
My MEX function generates errors.	You must recompile the MEX file from the source code. If using the <code>-R2017b</code> flag does not resolve the issue, then there might be incompatibilities in your source code. For information about incompatibilities, see “Can I Run Existing MEX Functions?” on page 7-45 MathWorks recommends that you update your MEX source code to use the interleaved complex API.

See Also

More About

- “MATLAB Support for Interleaved Complex API in MEX Functions” on page 7-31
- “Upgrade MEX Files to Use Interleaved Complex API” on page 7-36
- “Troubleshooting MEX API Incompatibilities” on page 7-48

Troubleshooting MEX API Incompatibilities

File Is Not A MEX File

For more information, see “MEX Platform Compatibility” on page 7-70.

MEX File Compiled With Incompatible Options

When you build object files into a MEX function, make sure they are built with the same version of the C or Fortran Matrix API.

This error occurs when you compile two or more files independently with the `-c` compile-only option, then try to build them into a MEX function. For example:

```
mex -c function1.c -largeArrayDims
mex -c function2.c -R2018a
mex function1.o function2.o
```

MEX File Compiled With One API And Linked With Another

This error occurs when you compile a file with the `-c` compile-only option and then link with a version of the API that is incompatible. For example, if you use the following commands to build a MEX file, then the function errors at runtime.

```
mex -c function1.c -largeArrayDims
mex function1.o -R2018a
```

C++ MEX File Using MATLAB Data API Compiled With Incompatible Option

If you create a C++ MEX file using functions in the “MATLAB Data API”, then the following build command errors.

```
mex function.cpp -R2017b
```

Use this command instead.

```
mex function.cpp
```

Custom-built MEX File Not Supported In Current Release

MATLAB does not find a version number in the MEX file. The MEX file uses functions in an API that requires a version number. For more information, see <https://www.mathworks.com/matlabcentral/answers/377799-compiling-mex-files-without-the-mex-command>.

MEX File Is Compiled With Outdated Option

Your source code is compatible with the interleaved complex API. For best results, replace the `mex -largeArrayDims` build option with the `-R2018a` option.

MEX File Calls An Untyped Data Access Function

For more information, see “Typed Data Access in C MEX Files” on page 8-65.

MEX File Calls A 32-bit Function

For more information, see “Upgrade MEX Files to Use 64-Bit API” on page 7-51.

MEX File Does Not Contain An Entry Point

For more information, see “MEX Platform Compatibility” on page 7-70.

MEX File Built In MATLAB Release Not Supported In Current Release

For more information, see “MEX Version Compatibility” on page 7-77.

See Also

More About

- “Upgrade MEX Files to Use Interleaved Complex API” on page 7-36

- “Do I Need to Upgrade My MEX Files to Use Interleaved Complex API?” on page 7-45

Upgrade MEX Files to Use 64-Bit API

The `mex` command uses the `-largeArrayDims` option by default. This topic describes how to upgrade your MEX files to use the 64-bit API.

You can continue to use the 32-bit API by calling the `mex` command with the `-compatibleArrayDims` option. However, for more information about using this option, see “What If I Do Not Upgrade?” on page 7-59.

To review and update MEX file source code, use the following checklist.

- 1 Prepare your code before editing — see “Back Up Files and Create Tests” on page 7-52.
- 2 Iteratively change and test code.

Before building your MEX files with the 64-bit API, refactor your existing code by checking for the following conditions:

- a “Update Variables” on page 7-52.
- b “Replace Unsupported Functions” on page 7-55.
- c If necessary, “Upgrade Fortran MEX Files to use 64-bit API” on page 7-62.

After each change, build and test your code:

- Build with the 32-bit API. For example, to build `myMexFile.c`, type:

```
mex -compatibleArrayDims myMexFile.c
```

- Test after each refactoring — see “Test, Debug, and Resolve Differences After Each Refactoring Iteration” on page 7-55.

- 3 Compile using the 64-bit API. To build `myMexFile.c`, type:

```
mex myMexFile.c
```

- 4 Resolve failures and warnings — see “Resolve `-largeArrayDims` Build Failures and Warnings” on page 7-55.
- 5 Compare Results — see “Execute 64-Bit MEX File and Compare Results with 32-Bit Version” on page 7-56.
- 6 Check memory — see “Experiment with Large Arrays” on page 7-56.

The following procedures use C/C++ terminology and example code. Fortran MEX files share issues, with more tasks described in “Upgrade Fortran MEX Files to use 64-bit API” on page 7-62.

Back Up Files and Create Tests

Before modifying your code, verify that the MEX file works with the 32-bit API. At a minimum, build a list of expected inputs and outputs, or create a full test suite. Use these tests to compare the results with the updated source code. The results should be identical.

Back up all source, binary, and test files.

Update Variables

To handle large arrays, convert variables containing array indices or sizes to use the `mwSize` and `mwIndex` types instead of the 32-bit `int` type. Review your code to see if it contains the following types of variables:

- Variables used directly by the Matrix API functions — see “Update Arguments Used to Call Functions in the 64-Bit API” on page 7-52.
- Intermediate variables — see “Update Variables Used for Array Indices and Sizes” on page 7-53.
- Variables used as both size/index values and as 32-bit integers — see “Analyze Other Variables” on page 7-54.

Update Arguments Used to Call Functions in the 64-Bit API

Identify the 64-bit API functions in your code that use the `mwSize` / `mwIndex` types. For the list of functions, see “Using the 64-Bit API” on page 8-62. Search for the variables that you use to call the functions. Check the function signature, shown under the **Syntax** heading on the function reference documentation. The signature identifies the variables that take `mwSize` / `mwIndex` values as input or output values. Change your variables to use the correct type.

For example, suppose that your code uses the `mxCreateDoubleMatrix` function, as shown in the following statements:

```
int nrows,ncolumns;
...
y_out = mxCreateDoubleMatrix(nrows, ncolumns, mxREAL);
```

To see the function signature, type:

```
doc mxCreateDoubleMatrix
```

The signature is:

```
mxAArray *mxCreateDoubleMatrix(mwSize m, mwSize n,
    mxComplexity ComplexFlag)
```

The type for input arguments *m* and *n* is *mwSize*. Change your code as shown in the table.

Replace:	With:
<code>int nrows,ncolumns;</code>	<code>mwSize nrows,ncolumns;</code>

Update Variables Used for Array Indices and Sizes

If your code uses intermediate variables to calculate size and index values, use *mwSize* / *mwIndex* for these variables. For example, the following code declares the inputs to `mxCreateDoubleMatrix` as type *mwSize*:

```
mwSize nrows,ncolumns;    /* inputs to mxCreateDoubleMatrix */
int numDataPoints;
nrows = 3;
numDataPoints = nrows * 2;
ncolumns = numDataPoints + 1;
...
y_out = mxCreateDoubleMatrix(nrows, ncolumns, mxREAL);
```

This example uses the intermediate variable, *numDataPoints* (of type `int`), to calculate the value of *ncolumns*. If you copy a 64-bit value from *nrows* into the 32-bit variable, *numDataPoints*, the resulting value truncates. Your MEX file could crash or produce incorrect results. Use type *mwSize* for *numDataPoints*, as shown in the following table.

Replace:	With:
<code>int numDataPoints;</code>	<code>mwSize numDataPoints;</code>

Analyze Other Variables

You do not need to change every integer variable in your code. For example, field numbers in structures and status codes are of type `int`. However, you need to identify variables used for multiple purposes and, if necessary, replace them with multiple variables.

The following example creates a matrix, *myNumeric*, and a structure, *myStruct*, based on the number of sensors. The code uses one variable, *numSensors*, for both the size of the array and the number of fields in the structure.

```
mxArray *myNumeric, *myStruct;
int numSensors;
mwSize m, n;
char **fieldnames;
...
myNumeric = mxCreateDoubleMatrix(numSensors, n, mxREAL);
myStruct = mxCreateStructMatrix(m, n, numSensors, fieldnames);
```

The function signatures for `mxCreateDoubleMatrix` and `mxCreateStructMatrix` are:

```
mxArray *mxCreateDoubleMatrix(mwSize m, mwSize n,
                               mxComplexity ComplexFlag)
mxArray *mxCreateStructMatrix(mwSize m, mwSize n,
                               int nfields, const char **fieldnames);
```

For the `mxCreateDoubleMatrix` function, your code uses *numSensors* for the variable *m*. The type for *m* is `mwSize`. For the `mxCreateStructMatrix` function, your code uses *numSensors* for the variable *nfields*. The type for *nfields* is `int`. To handle both functions, replace *numSensors* with two new variables, as shown in the following table.

Replace:	With:
<code>int numSensors;</code>	<code>/* create 2 variables */ /* of different types */ mwSize numSensorSize; int numSensorFields;</code>
<code>myNumeric = mxCreateDoubleMatrix(numSensors, n, mxREAL);</code>	<code>/* use mwSize variable */ /* numSensorSize */ myNumeric = mxCreateDoubleMatrix(numSensorSize, n, mxREAL);</code>

Replace:	With:
<pre>myStruct = mxCreateStructMatrix(m, n, numSensors, fieldnames);</pre>	<pre>/* use int variable */ /* numSensorFields */ myStruct = mxCreateStructMatrix(m, n, numSensorFields, fieldnames);</pre>

Replace Unsupported Functions

While updating older MEX files, you could find calls to unsupported functions, such as `mxCreateFull`, `mxGetName`, or `mxIsString`. MATLAB removed support for these functions in Version 7.1 (R14SP3). You cannot use unsupported functions with 64-bit array dimensions. For the list of unsupported functions and the recommended replacements, see “Obsolete Functions No Longer Documented”.

Update your code to use an equivalent function, if available. For example, use `mxCreateDoubleMatrix` instead of `mxCreateFull`.

Test, Debug, and Resolve Differences After Each Refactoring Iteration

To build `myMexFile.c` with the 32-bit API, type:

```
mex -compatibleArrayDims myMexFile.c
```

Use the tests you created at the beginning of this process to compare the results of your updated MEX file with your original binary file. Both MEX files should return identical results. If not, debug and resolve any differences. Differences are easier to resolve now than when you build using the 64-bit API.

Resolve -largeArrayDims Build Failures and Warnings

After reviewing and updating your code, compile your MEX file using the large array handling API. To build `myMexFile.c` with the 64-bit API, type:

```
mex myMexFile.c
```

Since the `mwSize` / `mwIndex` types are MATLAB types, your compiler sometimes refers to them as `size_t`, `unsigned_int64`, or by other similar names.

Most build problems are related to type mismatches between 32-bit and 64-bit types. Refer to Step 5 in *How do I update MEX-files to use the large array handling API (-largeArrayDims)?* to identify common build problems for specific compilers, and possible solutions.

Execute 64-Bit MEX File and Compare Results with 32-Bit Version

Compare the results of running your MEX file compiled with the 64-bit API with the results from your original binary. If there are any differences or failures, use a debugger to investigate the cause. For information on the capabilities of your debugger, refer to your compiler documentation.

To identify issues—and possible solutions—you might encounter when running your MEX files, refer to Step 6 in *How do I update MEX-files to use the large array handling API (-largeArrayDims)?*.

After you resolve issues and upgrade your MEX file, it replicates the functionality of your original code while using the large array handling API.

Experiment with Large Arrays

If you have access to a machine with large amounts of memory, you can experiment with large arrays. An array of double-precision floating-point numbers (the default in MATLAB) with 2^{32} elements takes approximately 32 GB of memory.

For an example that demonstrates the use of large arrays, see the `arraySize.c` MEX file in “Handling Large mxArray in C MEX Files” on page 8-62.

See Also

Related Examples

- “Upgrade Fortran MEX Files to use 64-bit API” on page 7-62

- “Handling Large mxArray in C MEX Files” on page 8-62

More About

- “What If I Do Not Upgrade?” on page 7-59
- “Using the 64-Bit API” on page 8-62
- “Obsolete Functions No Longer Documented”

External Websites

- How do I update MEX-files to use the large array handling API (-largeArrayDims)?

MATLAB Support for 64-Bit Indexing

MATLAB Version 7.3 (R2006b) added support for 64-bit indexing. With 64-bit indexing, you can create variables with up to $2^{48}-1$ elements on 64-bit platforms. Before Version 7.3, the C/C++ and Fortran API Reference library functions used `int` in C/C++ and `INTEGER*4` in Fortran to represent array dimensions. These types limit the size of an array to 32-bit integers. Simply building and running MEX files on a 64-bit platform does not guarantee you access to the additional address space. You must update your MEX source code to take advantage of this functionality.

The following changes to the C Matrix API support 64-bit indexing:

- New types - `mwSize` and `mwIndex` - enabling large-sized data.
- Updated C Matrix API functions use `mwSize` and `mwIndex` types for inputs and outputs. These functions are called the 64-bit API or the large-array-handling API.

To help transition your MEX files to the 64-bit API, MATLAB maintains an interface, or compatibility layer. To build MEX files with this interface, use the `-compatibleArrayDims` flag.

Note Only variables representing array size or index value require the `mwSize` or `mwIndex` types. The C-language `int` data type is valid for variables representing, for example, the number of fields or arrays.

See Also

`mwIndex` | `mwSize`

Related Examples

- “Upgrade MEX Files to Use 64-Bit API” on page 7-51

More About

- C Matrix API

What If I Do Not Upgrade?

If you do not update your MEX source code, you can still build your MEX files using the `-compatibleArrayDims` option. Use this flag to ensure the desired behavior across versions of MATLAB. If you build without the `-compatibleArrayDims` flag, then one or more of the following could occur:

- Increased compiler warnings and/or errors from your native compiler
- Run-time errors
- Wrong answers

Can I Run Existing Binary MEX Files?

You can run existing binary MEX files without upgrading the files for use with the 64-bit API. However, incompatibilities might prevent execution of an existing MEX file. If your MEX file does not execute properly, search for `mex` in the relevant MATLAB release notes and review the Compatibility Considerations topics.

Must I Update Source MEX Files on 64-Bit Platforms?

If you build MEX files on 64-bit platforms or write platform-independent applications, you must upgrade your MEX files. To upgrade, review your source code, make appropriate changes, and rebuild using the `mex` command.

What action you take now depends on whether your MEX files currently use the 64-bit API. The following table helps you identify your next actions.

State of Your Source Code	Next Action
I do not plan to update my code.	You have chosen to opt out and you must build using the <code>-compatibleArrayDims</code> flag. However, in a future version of MATLAB, the compatibility layer, with the <code>-compatibleArrayDims</code> flag, might be unsupported.
I want to update my code. Where do I start?	See “Upgrade MEX Files to Use 64-Bit API” on page 7-51.

State of Your Source Code	Next Action
I use MEX files, but do not have access to the source code.	Ask the owner of the source code to follow the steps in “Upgrade MEX Files to Use 64-Bit API” on page 7-51.
I use third-party libraries.	<p>Ask the vendor if the libraries support 64-bit indexing. If not, you cannot use these libraries to create 64-bit MEX files. Build your MEX file using the <code>-compatibleArrayDims</code> flag.</p> <p>If the libraries support 64-bit indexing, review your source code, following the steps in “Upgrade MEX Files to Use 64-Bit API” on page 7-51, and then test.</p>
I updated my code in a previous release.	No change required. However, you no longer need to use the <code>-largeArrayDims</code> option when building the MEX file.
My MEX file generates errors.	You must recompile the MEX file from the source code. If using the <code>-compatibleArrayDims</code> flag does not resolve the issue, then there might be incompatibilities in your source code. For information about incompatibilities, see “Can I Run Existing Binary MEX Files?” on page 7-59. MathWorks recommends that you update your MEX source code to use the 64-bit API.

See Also

Related Examples

- “Upgrade MEX Files to Use 64-Bit API” on page 7-51

More About

- “MATLAB Support for 64-Bit Indexing” on page 7-58

- “MEX Version Compatibility” on page 7-77

Upgrade Fortran MEX Files to use 64-bit API

The steps in “Upgrade MEX Files to Use 64-Bit API” on page 7-51 apply to Fortran and C/C++ source files. Fortran uses similar API signatures, identical `mwSize / mwIndex` types, and similar compilers and debuggers.

However, to make your Fortran source code 64-bit compatible, perform these additional tasks.

Use Fortran API Header File

To make your Fortran MEX file compatible with the 64-bit API, use the `fintrf.h` header file in your Fortran source files. Name your source files with an uppercase `.F` file extension. For more information about these requirements, see “Components of Fortran MEX File” on page 10-2.

Declare Fortran Pointers

Pointers are 32-bit or 64-bit addresses, based on machine type. This requirement is not directly tied to array dimensions, but you could encounter problems when moving 32-bit code to 64-bit machines as part of this conversion.

For more information, see “Preprocessor Macros” on page 10-5 and `mwPointer`.

The C/C++ compiler automatically handles pointer size. In Fortran, MATLAB uses the `mwPointer` type to handle this difference. For example, `mxCreatDoubleMatrix` returns an `mwPointer`:

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

Require Fortran Type Declarations

Fortran uses implicit type definitions. Undeclared variables starting with letters `I` through `N` are implicitly declared type `INTEGER`. Variable names starting with other letters are implicitly declared type `REAL*4`. Using the implicit `INTEGER` type could work for 32-bit indices, but is not safe for large array dimension MEX files. To force you to declare all variables, add the `IMPLICIT NONE` statement to your Fortran subroutines. For example:


```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
implicit none
```

This statement helps identify 32-bit integers in your code that do not have explicit type declarations. Then, you can declare them as `INTEGER*4` or `mwSize / mwIndex`, as appropriate. For more information on `IMPLICIT NONE`, refer to your Fortran compiler documentation.

Use Variables in Function Calls

If you use a number as an argument to a function, your Fortran compiler could assign the argument an incorrect type. On a 64-bit platform, an incorrect type can produce `Out of Memory` errors, segmentation violations, or incorrect results. For example, definitions for the argument types for the `mxCreateDoubleMatrix` function are:

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

Suppose that you have a C/C++ MEX file with the following statement:

```
myArray = mxCreateDoubleMatrix(2, 3, mxREAL);
```

Most C/C++ compilers interpret the number 2 as a 64-bit value. Some Fortran compilers cannot detect this requirement, and supply a 32-bit value. For example, an equivalent Fortran statement is:

```
myArray = mxCreateDoubleMatrix(2, 3, 0)
```

The compiler interprets the value of the `ComplexFlag` argument 0 correctly as type `INTEGER*4`. However, the compiler could interpret the argument 2 as a 32-bit value, even though the argument `m` is declared type `mwSize`.

A compiler-independent solution to this problem is to declare and use an `mwSize / mwIndex` variable instead of a literal value. For example, the following statements unambiguously call the `mxCreateDoubleMatrix` function in Fortran:

```
mwSize nrows, ncols
INTEGER*4 flag
nrows = 2
ncols = 3
flag = 0
myArray = mxCreateDoubleMatrix(nrows, ncols, flag)
```

Manage Reduced Fortran Compiler Warnings

Some Fortran compilers cannot detect as many type mismatches as similar C/C++ compilers. This inability can complicate the step “Resolve -largeArrayDims Build Failures and Warnings” on page 7-55 by leaving more issues to find with your debugger in the step “Execute 64-Bit MEX File and Compare Results with 32-Bit Version” on page 7-56.

See Also

[mwIndex](#) | [mwPointer](#) | [mwSize](#)

Related Examples

- “Upgrade MEX Files to Use 64-Bit API” on page 7-51

More About

- “Components of Fortran MEX File” on page 10-2
- “Preprocessor Macros” on page 10-5

External Websites

- How do I update MEX-files to use the large array handling API (-largeArrayDims)?

Upgrade MEX Files to Use Graphics Objects

MATLAB Version 8.4 (R2014b) changes the data type of handles to graphics objects from `double` to *object*.

Before Version 8.4, MEX files used the C/C++ and Fortran API Reference library functions `mexGet` and `mexSet`, which declare the input handle argument as type `double`. If your MEX function uses `mexGet` or `mexSet`, MATLAB displays the following error.

```
Error using mex
Deprecated MEX function mexGet|mexSet was called. Either update the source code
to use mxGetProperty|mxSetProperty, OR rerun MEX with the -DMEX_DOUBLE_HANDLE
added to the command line to enter compatibility mode.
```

To upgrade your MEX file, consider one or more of the following actions.

Replace `mexGet` and `mexSet` Functions

To upgrade a MEX file to use a graphics object, replace calls to `mexGet` with `mxGetProperty` and calls to `mexSet` with `mxSetProperty`. The following program listings show an example of a before and after source MEX file.

The following code uses `mexCallMATLAB` to create a plot, which returns the graphics handle in variable `plhs[0]`. To change the line color, the example uses `mxGetScalar` to convert the handle to a `double`, then passes it to `mexGet` and `mexSet`.

```
#include "mex.h"
#define RED 0
#define GREEN 1
#define BLUE 2

void fill_array(double *x)
{
    int i = 0;
    for(i = 0 ; i < 4 ; i++)
    {
        x[i] = i+1;
    }
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    mxArray *color;
    int ret;
```

```
double handle;
mxArray *copycolor;
double *acolor;

mxArray *data = mxCreateDoubleMatrix(1,4,mxREAL);
fill_array(mxGetPr(data));

ret = mexCallMATLAB(1,&plhs[0],1,&data,"plot");
if(!ret)
{
    handle = mxGetScalar(plhs[0]);
    color = mexGet(handle,"Color");
    copycolor = mxDuplicateArray(color);
    acolor = mxGetPr(copycolor);
    acolor[RED] = (1 + acolor[RED]) /2;
    acolor[GREEN] = acolor[GREEN]/2;
    acolor[BLUE] = acolor[BLUE]/2;

    mexSet(handle,"Color",copycolor);
    mxSetProperty(plhs[0],0,"Color",copycolor);
}
}
```

When you build this MEX file, MATLAB displays an error message.

To change the source file, make the following edits. This code uses the variable `plhs[0]` in `mxGetProperty` to get the `Color` property directly. There is no need to create an intermediate `handle` variable.

```
#include "mex.h"
#define RED 0
#define GREEN 1
#define BLUE 2

void fill_array(double *x)
{
    int i = 0;
    for(i = 0 ; i < 4 ; i++)
    {
        x[i] = i+1;
    }
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
```

```

mxArray *color;
int ret;

mxArray *copycolor;
double *acolor;

mxArray *data = mxCreateDoubleMatrix(1,4,mxREAL);
fill_array(mxGetPr(data));

ret = mexCallMATLAB(1,&plhs[0],1,&data,"plot");
if(!ret)
{
    color = mxGetProperty(plhs[0],0,"Color");
    copycolor = mxDuplicateArray(color);
    acolor = mxGetPr(copycolor);
    acolor[RED] = (1 + acolor[RED]) /2;
    acolor[GREEN] = acolor[GREEN]/2;
    acolor[BLUE] = acolor[BLUE]/2;

    mxSetProperty(plhs[0],0,"Color",copycolor);
}
}

```

To build this MEX file, type:

```
mex mymex.c
```

```
Building with 'Microsoft Visual C++ 2012 (C)'.
MEX completed successfully.
```

Alternatively, you can build the original source file by following the steps in “I Want to Rebuild MEX Source Code Files” on page 7-68.

mex Automatically Converts Handle Type

If your MEX function uses the `mexCallMATLAB` or `mexGetVariable` functions to get a graphics handle and to pass the handle to the `mexGet` and `mexSet` APIs, then MATLAB automatically detects that behavior and your MEX function continues to execute correctly. You know that your MEX function uses this pattern if the function executes without error.

If you rebuild this MEX file in MATLAB R2014b or later, MATLAB displays an error message. To rebuild the file, follow the instructions in either “Replace `mexGet` and `mexSet` Functions” on page 7-65 or “I Want to Rebuild MEX Source Code Files” on page 7-68.

I Want to Rebuild MEX Source Code Files

If you rebuild your MEX source files in MATLAB R2014b or later, MATLAB displays an error message.

You might be able to use the `mex` command compatibility flag, `-DMEX_DOUBLE_HANDLE`, to build the MEX file to work with graphics objects. If the MEX function calls a function that returns a graphics handle using the `mexCallMATLAB` or `mexGetVariable` functions, MATLAB automatically detects and converts the handle type. To build the source file, `mymex.c`, type:

```
mex -DMEX_DOUBLE_HANDLE mymex.c
```

If you pass a graphics handle to a MEX function, convert the handle to `double` before calling the function. For more information, see “I Do Not Have MEX Source Code File” on page 7-68.

I Do Not Have MEX Source Code File

If you get a run-time error and you do not have the source code, you might be able to use the following workaround. Use this workaround only for MEX functions that take a graphics handle as an input argument.

Before you pass a graphics handle to the MEX function, first convert the handle to a `double`. For example, if you call MEX function `mymex`:

```
Y = 1:10;  
h = plot(Y);  
mymex(h)
```

then add a statement to convert the handle `h` to `double`:

```
Y = 1:10;  
h = plot(Y);  
h = double(h);  
mymex(h)
```

See Also

`mxGetProperty` | `mxSetProperty`

More About

- “Graphics Object Handles”

MEX Platform Compatibility

If you obtain a binary MEX file from another source, be sure that the file was compiled for the same platform on which you want to run it. The file extension reflects the platform, as shown in this table. To determine the extension for your platform, use the `mexext` function.

MEX File Platform-Dependent Extension

Platform	Binary MEX File Extension
Linux (64-bit)	<code>mexa64</code>
Apple Mac (64-bit)	<code>mexmaci64</code>
Windows (64-bit)	<code>mexw64</code>

Note It is not possible to run a 32-bit MEX file from within a 64-bit version of MATLAB.

See Also

`mexext`

More About

- “MEX Version Compatibility” on page 7-77

Invalid MEX File Errors

If MATLAB cannot find all `.dll` files referenced by a MEX file, it cannot load the MEX file. For example, suppose MEX file A depends on library `subA.dll` which is not on the system path. If you call `A.mexw64` in folder `L:\mylibs\`, MATLAB displays:

```
Invalid MEX-file 'L:\mylibs\A.mexw64':  
Missing dependent shared libraries  
'subA.dll' required by 'L:\mylibs\A.mexw64'
```

If a symbol required by a shared library is missing, MATLAB displays a message similar to the following:

```
Invalid MEX-file 'L:\mylibs\A.mexw64':  
Missing symbol 'q' in 'L:\mylibs\subA.dll' required by L:\mylibs\A.mexw64'
```

To find library dependencies:

- On Windows systems, download the Dependency Walker utility from the website <http://www.dependencywalker.com>.
- On Linux systems, use:

```
ldd -d libname.so
```
- On macOS systems, use:

```
otool -L libname.dylib
```

For `.dll` files that the MEX file linked against when it was built, the `.dll` files must be on the system path or in the same folder as the MEX file.

MEX files might require additional libraries that are not linked to the MEX file. Failure to find one of these explicitly loaded libraries might not prevent a MEX file from loading, but prevents it from working correctly. The code that loads the libraries controls the search path used to find these libraries. The search path might not include the folder that contains the MEX file. Consult the library documentation on proper installation locations.

Possible reasons for failure include:

- MATLAB version incompatibility. For more information, see “MEX Version Compatibility” on page 7-77.
- Missing compiler run-time libraries. If your system does not have the same compiler that built the MEX file, see the Microsoft MSDN website for information about Visual C++® Redistributable Packages.

- Missing or incorrectly installed specialized run-time libraries. Contact your MEX file or library vendor.

See Also

More About

- “MEX Version Compatibility” on page 7-77

External Websites

- How do I determine which libraries my MEX-file or stand-alone application requires?

Run MEX File You Receive from Someone Else

To call a MEX file, put the file on your MATLAB path. Then type the name of the file, without the file extension.

If you have MEX file source code, see “Build C MEX Function” on page 7-16 for information about creating the executable function.

If you get run-time errors when you call a MEX file that you did not create, consider the following:

- “MEX Platform Compatibility” on page 7-70
- “MEX Version Compatibility” on page 7-77
- On Windows platforms, install the C++ compiler run-time libraries used to create the MEX file. This step is needed if you do not have the same compiler installed on your machine that was used to compile the MEX file.
- If the MEX file uses specialized run-time libraries, then those libraries must be installed on your system.

If you write a MEX file, build it, and then execute it in the same MATLAB session, all the dependent libraries are available, as expected. However, if you receive a MEX file from another MATLAB user, you might not have all the dependent libraries.

A MEX file is a dynamically linked subroutine that the MATLAB interpreter loads and executes when you call the function. Dynamic linking means that when you call the function, the program looks for dependent libraries. MEX files use MATLAB run-time libraries and language-specific libraries. A MEX file might also use specialized run-time libraries. The code for these libraries is not included in the MEX file; the libraries must be present on your computer when you run the MEX file.

For troubleshooting library dependencies, see “Invalid MEX File Errors” on page 7-71.

For information about how MATLAB finds a MEX file, see “Files and Folders that MATLAB Accesses”.

MEX File Dependent Libraries

When you build a MEX file, MATLAB dynamically links your code with the following libraries:

- MATLAB run-time libraries `libmex.dll` and `libmx.dll`. To avoid compatibility issues, run the MEX file with the same version of MATLAB that was used to create the MEX file. For more information, see “MEX Version Compatibility” on page 7-77.
- Language-specific libraries, provided by the compiler. C-language MEX files built on Windows systems require Visual C++ run-time libraries.
- Other run-time libraries, which you specify in the build command.

Dynamic linking means that when you call the function, the program looks for these dependent libraries. The code for these libraries is not included in the MEX file. When you share a MEX file with another MATLAB user, these libraries must be present on the user’s computer.

Document Build Information in the MEX File

This example shows how to document the `xtimesy` MEX file built on a Windows platform using a Microsoft Visual C++ compiler.

When you share a MEX file, your users need the following information about the configuration used to build the MEX file:

- MATLAB version.
- Build platform.
- Compiler.

Copy the source file to a folder on your MATLAB path.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'refbook', 'xtimesy.c'), '.')
```

Create a help file, `xtimesy.m`, and copy the header information from the source file.

```
% xtimesy.m Help file for XTIMESY MEX file
%
% XTIMESY Multiplies a scalar and a matrix
%   C = XTIMESY(b,A) multiplies scalar b with matrix A,
%   and returns the result in C
%
%   MEX File function.
```

Identify your MATLAB version.

```
v = ver('matlab');
v.Release
```

```
ans =
(R2012a)
```

Identify your platform.

```
archstr = computer('arch')
```

```
archstr =
win64
```

Identify the MEX file extension.

```
ext = mexext
```

```
ext =  
mexw64
```

Identify your C compiler.

```
cc = mex.getCompilerConfigurations('C', 'Selected');  
cc.Name
```

```
ans =  
Microsoft Visual C++ 2008 (C)
```

Add this information to the help file.

```
% xtimesy.m Help file for XTIMESY MEX file  
%  
% XTIMESY Multiplies a scalar and a matrix  
% C = XTIMESY(b,A) multiplies scalar b with matrix A,  
% and returns the result in C  
%  
% Created with:  
% MATLAB R2012a  
% Platform: win64  
% Microsoft Visual C++ 2008  
  
% MEX File function.
```

Provide your users with the following.

- `xtimesy.mexw64`
- `xtimesy.m`
- Instructions for downloading and installing the correct Microsoft Visual C++ run-time library.
- If you build a MEX file with a third-party library, instructions for acquiring and installing the necessary files.

See Also

Related Examples

- “Use Help Files with MEX Functions” on page 7-5

MEX Version Compatibility

For best results, your version of MATLAB must be the same version that was used to create the MEX file.

MEX files use MATLAB run-time libraries. A MEX file that was created on an earlier version of MATLAB usually runs on later versions of MATLAB. If the MEX file generates errors, recompile the MEX file from the source code.

Sometimes a MEX file created on a newer version of MATLAB runs on an older version of MATLAB, but this is not supported.

See Also

More About

- “MEX Platform Compatibility” on page 7-70

Getting Help When MEX Fails

To help diagnose compiler set up and build errors, call `mex` with the verbose option, `-v`. For an example of the information `mex` provides, type the following commands from a writable folder:

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'refbook', 'timestwo.c'), '.', 'f')
mex -v timestwo.c
```

Errors Finding Supported Compiler

In verbose mode, `mex` displays the steps used to find a supported compiler and to determine if it is properly installed. Each step begins with the following text:

```
... Looking for
```

If the compiler is not configured properly, these messages show you the expected values for specific files, paths, and variables in the configuration.

If the compiler is found, `mex` displays a message similar to:

```
Building with 'Microsoft Visual C++ 2012 (C)'.
```

Errors Building MEX Function

After locating the installed compiler, indicated by the “Building with” message, verbose mode displays the compile and link commands `mex` passes to the build tools. For example, the compile command on Windows platforms might be similar to the following:

```
cl /c /GR /W3 /EHs /nologo /MD /DMX_COMPAT_32
/D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE
-I"matlabroot\extern\include" -I"matlabroot\simulink\include"
/02 /Oy- /DNDEBUG C:\work\mex\timestwo.c /FoC:\work\timestwo.obj
timestwo.c
```

`mex` displays error messages from the compiler build tools. For information about errors and warnings, see your compiler or language reference documentation.

If you have experience with program development and want to modify a command parameter, use the `mex varname=varvalue` option.

Preview mex Build Commands

To display the build command details without executing the commands, type:

```
mex -n timestwo.c
```

See Also

mex

MEX API Is Not Thread Safe

Do not call a single session of MATLAB on separate threads from a MEX file. The MEX and Matrix Library APIs are not multi-threaded.

You can create threads from a C MEX file; however, accessing MATLAB from those threads is not supported. Do not call any MEX API functions from the spawned threads, including `printf`, which is defined as `mexPrintf` in the `mex.h` header file.

For more information, see [Is it possible to start new threads from a C-MEX file?](#) in MATLAB Answers.

Compiling MEX File Fails

Build Example Files

Can you compile and run the `timestwo.c` or `timestwo.f` example files? See “Build C MEX Function” on page 7-16 or “Build Fortran MEX File” on page 10-13.

Use Supported Compiler

Are you using a supported compiler? For an up-to-date list of supported compilers, see Supported and Compatible Compilers.

File Not Found on Windows

The `mex` command cannot find files located in folder names that contain non-ASCII characters.

Linux gcc -fPIC Errors

If you link a static library with a MEX file, which is a shared library, you might get an error message containing the text `recompile with -fPIC`. Try compiling the static library with the `-fPIC` flag to create position-independent code. For information about using the `gcc` compiler, see <https://www.gnu.org/>. For an up-to-date list of supported compilers, see Supported and Compatible Compilers.

Compiler Errors in Fortran MEX Files

When you compile a Fortran MEX file using a free source form format, MATLAB displays an error message of the following form:

```
Illegal character in statement label field
```

`mex` supports the fixed source form. For information about the difference between free and fixed source forms, refer to a FORTRAN 77 Language Reference manual.

Syntax Errors Compiling C/C++ MEX Files on UNIX

If MATLAB header files generate multiple syntax errors when you compile your code on UNIX systems, you might be using a non-ANSI C compiler.

The most common configuration problem creating C/C++ MEX files on UNIX systems involves using a non-ANSI C compiler, or failing to pass a compiler flag telling it to compile ANSI C code.

One way to know if you have this type of configuration problem is if the MATLAB header files generate multiple syntax errors when you compile your code. If necessary, obtain an ANSI C compiler.

See Also

Related Examples

- “Build C MEX Function” on page 7-16
- “Build Fortran MEX File” on page 10-13

External Websites

- Supported and Compatible Compilers

Symbol mexFunction Unresolved or Not Defined

Attempting to compile a MEX function that does not include a gateway function generates errors about the `mexFunction` symbol. For example, using a C/C++ compiler, MATLAB displays information like:

```
LINK : error LNK2001: unresolved external symbol mexFunction
```

Using a Fortran compiler, MATLAB displays information like:

```
unresolved external symbol _MEXFUNCTION
```

On macOS platforms, MATLAB displays information like:

```
Undefined symbols for architecture x86_64:  
"_mexfunction_", referenced from:  
-exported_symbol[s_list] command line option
```

See Also

More About

- “Components of C MEX File” on page 8-2

MEX File Segmentation Fault

If a binary MEX file causes a segmentation violation or assertion, it means that the MEX file attempted to access protected, read-only, or unallocated memory.

These types of programming errors can be difficult to track down. Segmentation violations do not always occur at the same point as the logical errors that cause them. If a program writes data to an unintended section of memory, an error might not occur until the program reads and interprets the corrupted data. Therefore, a segmentation violation can occur after the MEX file finishes executing.

One cause of memory corruption is to pass a null pointer to a function. To check for this condition, add code in your MEX file to check for invalid arguments to MEX Library and Matrix API functions.

To troubleshoot problems of this nature, run MATLAB within a debugging environment.

See Also

Related Examples

- “Debug on Mac Platforms” on page 8-57
- “Debug on Linux Platforms” on page 8-55
- “Debug Fortran MEX Files” on page 10-23

MEX File Generates Incorrect Results

If your program generates the wrong answers, consider the following.

- Check for errors in the computational logic.
- Check if the program reads from an uninitialized section of memory. For example, reading the 11th element of a 10-element vector yields unpredictable results.
- Check if the program overwrites valid data due to memory mishandling. For example, writing to the 15th element of a 10-element vector overwrites data in the adjacent variable in memory. This case can be handled in a similar manner as segmentation violations.

In all these cases, you can use `mexPrintf` to examine data values at intermediate stages. Alternatively, run MATLAB within a debugger.

See Also

`mexPrintf`

Related Examples

- “Debug on Mac Platforms” on page 8-57
- “Debug on Linux Platforms” on page 8-55
- “Debug Fortran MEX Files” on page 10-23

More About

- “MEX File Segmentation Fault” on page 7-84

Memory Management Issues

Overview

Note The examples in this topic use functions in the interleaved complex API. To build applications with these functions, call `mex` with the release-specific option `-R2018a`.

When a MEX function returns control to MATLAB, it returns the results of its computations in the output arguments—the `mxArrays` contained in the left-side arguments `plhs[]`. These arrays must have a temporary scope, so do not pass arrays created with the `mexMakeArrayPersistent` function in `plhs`. MATLAB destroys any `mxArray` created by the MEX function that is not in `plhs`. MATLAB also frees any memory that was allocated in the MEX function using the `mxMalloc`, `mxRealloc`, or `mxRealloc` functions.

In general, MathWorks® recommends that MEX 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. This approach is consistent with other MATLAB API applications (MAT-file applications, engine applications, and MATLAB Compiler generated applications, which do not have any automatic cleanup mechanism.)

However, do not destroy an `mxArray` in a source MEX file when it is:

- passed to the MEX file in the right-hand side list `prhs[]`
- returned in the left side list `plhs[]`
- returned by `mexGetVariablePtr`
- used to create a structure

This section describes situations specific to memory management. We recommend that you review code in your source MEX files to avoid using these functions in the following situations. For more information, see “Automatic Cleanup of Temporary Arrays in MEX Files” on page 8-69 and “Persistent `mxArrays`” on page 8-67. For guidance on memory issues, see “Strategies for Efficient Use of Memory”.

Potential memory management problems include:

Improperly Destroying an mxArray

Do not use `mxFree` to destroy an `mxArray`.

Example

In the following example, `mxFree` does not destroy the array object. This operation frees the structure header associated with the array, but MATLAB still operates as if the array object needs to be destroyed. Thus MATLAB tries to destroy the array object, and in the process, attempts to free its structure header again:

```
mxArray *temp = mxCreateDoubleMatrix(1,1,mxREAL);
...
mxFree(temp); /* INCORRECT */
```

Solution

Call `mxDestroyArray` instead:

```
mxDestroyArray(temp); /* CORRECT */
```

Incorrectly Constructing a Cell or Structure mxArray

Do not call `mxSetCell` or `mxSetField` variants with `prhs[]` as the member array.

Example

In the following example, when the MEX file returns, MATLAB destroys the entire cell array. Since this includes the members of the cell, this implicitly destroys the input arguments of the MEX file. This can cause several strange results, generally having to do with the corruption of the caller workspace, if the right-hand side argument used is a temporary array (for example, a literal or the result of an expression):

```
myfunction('hello')
/* myfunction is the name of your MEX file and your code
/* contains the following: */

    mxArray *temp = mxCreateCellMatrix(1,1);
    ...
    mxSetCell(temp, 0, prhs[0]); /* INCORRECT */
```

Solution

Make a copy of the right-hand side argument with `mxDuplicateArray` and use that copy as the argument to `mxSetCell` (or `mxSetField` variants). For example:

```
mxSetCell(temp, 0, mxDuplicateArray(prhs[0])); /* CORRECT */
```

Creating a Temporary mxArray with Improper Data

Do not call `mxDestroyArray` on an mxArray whose data was not allocated by an API routine.

Example

If you call `mxSetDoubles`, `mxSetComplexDoubles`, or any of the typed data access functions specifying memory that was not allocated by `mxMalloc`, `mxRealloc`, or `mxRealloc` as the intended data block (second argument), then when the MEX file returns, MATLAB attempts to free the pointers to real data and imaginary data (if any). Thus MATLAB attempts to free memory, in this example, from the program stack:

```
mxArray *temp = mxCreateDoubleMatrix(0,0,mxREAL);
double data[5] = {1,2,3,4,5};
...
mxSetM(temp,1); mxSetN(temp,5); mxSetDoubles(temp, data);
/* INCORRECT */
```

Solution

Rather than use `mxSetDoubles` to set the data pointer, instead, create the mxArray with the right size and use `memcpy` to copy the stack data into the buffer returned by `mxGetDoubles`:

```
mxArray *temp = mxCreateDoubleMatrix(1,5,mxREAL);
double data[5] = {1,2,3,4,5};
...
memcpy(mxGetDoubles(temp), data, 5*sizeof(double)); /* CORRECT */
```

Creating Potential Memory Leaks

Before Version 5.2, if you created an mxArray using one of the API creation routines and then you overwrote the pointer to the data using `mxSetDoubles`, MATLAB still freed the original memory. MATLAB no longer frees the memory.

For example:

```
pr = mxCalloc(5*5, sizeof(double));
... <load data into pr>
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
mxSetDoubles(plhs[0], pr); /* INCORRECT */
```

now leaks 5*5*8 bytes of memory, where 8 bytes is the size of a double.

You can avoid that memory leak by changing the code to:

```
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
pr = mxGetDoubles(plhs[0]);
... <load data into pr>
```

or alternatively:

```
pr = mxCalloc(5*5, sizeof(double));
... <load data into pr>
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
mxFree(mxGetDoubles(plhs[0]));
mxSetDoubles(plhs[0], pr);
```

The first solution is more efficient.

Similar memory leaks can also occur when using `mxSetDoubles`, `mxSetComplexDoubles`, `mxSetIr`, `mxSetJc`, or any of the numeric typed data access functions. You can avoid memory leaks by changing the code as described in this section.

Improperly Destroying a Structure

For a structure, you must call `mxDestroyArray` only on the structure, not on the field data arrays. A field in the structure points to the data in the array used by `mxSetField` or `mxSetFieldByNumber`. When `mxDestroyArray` destroys the structure, it attempts to traverse down through itself and free all other data, including the memory in the data arrays. If you call `mxDestroyArray` on each data array, the same memory is freed twice which can corrupt memory.

Example

The following example creates three arrays: one structure array `aStruct` and two data arrays, `myDataOne` and `myDataTwo`. Field name `one` contains a pointer to the data in `myDataOne`, and field name `two` contains a pointer to the data in `myDataTwo`.

```
mxArray *myDataOne;
mxArray *myDataTwo;
mxArray *aStruct;
const char *fields[] = { "one", "two" };

myDataOne = mxCreateDoubleScalar(1.0);
myDataTwo = mxCreateDoubleScalar(2.0);

aStruct = mxCreateStructMatrix(1,1,2,fields);
mxSetField( aStruct, 0, "one", myDataOne );
mxSetField( aStruct, 1, "two", myDataTwo );
mxDestroyArray(myDataOne);
mxDestroyArray(myDataTwo);
mxDestroyArray(aStruct); /* tries to free myDataOne and myDataTwo */
```

Solution

The command `mxDestroyArray(aStruct)` destroys the data in all three arrays:

```
...
aStruct = mxCreateStructMatrix(1,1,2,fields);
mxSetField( aStruct, 0, "one", myDataOne );
mxSetField( aStruct, 1, "two", myDataTwo );
mxDestroyArray(aStruct);
```

Destroying Memory in a C++ Class Destructor

Do not use the `mxFree` or `mxDestroyArray` functions in a C++ destructor of a class used in a MEX-function. If the MEX-function throws an error, MATLAB cleans up MEX-file variables, as described in “Automatic Cleanup of Temporary Arrays in MEX Files” on page 8-69.

If an error occurs that causes the object to go out of scope, MATLAB calls the C++ destructor. Freeing memory directly in the destructor means both MATLAB and the destructor free the same memory, which can corrupt memory.

See Also

`mxDestroyArray` | `mxFree`

More About

- “Strategies for Efficient Use of Memory”
- “Automatic Cleanup of Temporary Arrays in MEX Files” on page 8-69

C/C++ MEX-Files

Components of C MEX File

This topic is about MEX files built with the “C Matrix API”. For information about building C++ MEX files, see “Choosing MEX Applications” on page 7-2.

mexFunction Gateway Routine

The gateway routine is the entry point to the MEX file. It is through this routine that MATLAB accesses the rest of the routines in your MEX files. The name of the gateway routine is `mexFunction`. It takes the place of the `main` function in your source code.

Naming the MEX File

The name of the source file containing `mexFunction` is the name of your MEX file, and, hence, the name of the function you call in MATLAB.

The file extension of the binary MEX file is platform-dependent. You find the file extension using the `mexext` function, which returns the value for the current machine.

Required Parameters

The signature for `mexFunction` is:

```
void mexFunction(  
    int nlhs, mxArray *plhs[],  
    int nrhs, const mxArray *prhs[])
```

Place this function after your computational routine and any other functions in your source file.

This table describes the parameters for `mexFunction`.

Parameter	Description
<code>prhs</code>	Array of right-side input arguments.
<code>plhs</code>	Array of left-side output arguments.
<code>nrhs</code>	Number of right-side arguments, or the size of the <code>prhs</code> array.
<code>nlhs</code>	Number of left-side arguments, or the size of the <code>plhs</code> array.

Declare `prhs` and `plhs` as type `mxArray *`, which means they point to MATLAB arrays. They are vectors that contain pointers to the arguments of the MEX file. The keyword `const`, which modifies `prhs`, means that your MEX file does not modify the input arguments.

You can think of the name `prhs` as representing the “parameters, right-hand side,” that is, the input parameters. Likewise, `plhs` represents the “parameters, left-hand side,” or output parameters.

Managing Input and Output Parameters

Input parameters (found in the `prhs` array) are read-only; do not modify them in your MEX file. Changing data in an input parameter can produce undesired side effects.

You also must take care when using an input parameter to create output data or any data used locally in your MEX file. To copy an input array into a locally defined variable, `myData`, call the `mxDuplicateArray` function to make a copy of the input array. For example:

```
mxArray *myData = mxCreateStructMatrix(1,1,nfields,fnames);
mxSetField(myData,0,"myFieldName",mxDuplicateArray(prhs[0]));
```

For more information, see the troubleshooting topic “Incorrectly Constructing a Cell or Structure `mxArray`” on page 7-87.

Validating Inputs

For a list of functions to validate inputs to your functions, see the C Matrix API.

The `mxIsClass` function is a general-purpose way to test an `mxArray`. For example, suppose your second input argument (identified by `prhs[1]`) must be a full matrix of real numbers. To check this condition, use the following statements.

```
if(mxIsSparse(prhs[1]) ||
    mxIsComplex(prhs[1]) ||
    mxIsClass(prhs[1],"char")) {
    mexErrMsgIdAndTxt("MyToolbox:myfnc:nrhs",
        "input2 must be full matrix of real values.");
}
```

This example is not an exhaustive check. You can also test for structures, cell arrays, function handles, and MATLAB objects.

Computational Routine

The computational routine contains the code for performing the computations you want implemented in the binary MEX file. Although not required, consider writing the gateway routine, `mexFunction`, to call a computational routine. Use the `mexFunction` code as a wrapper to validate input parameters and to convert them into the types required by the computational routine.

If you write separate gateway and computational routines, you can combine them into one source file or into separate files. If you use separate files, the file containing `mexFunction` must be the first source file listed in the `mex` command.

See Also

`matlab::mex::ArgumentList` | `matlab::mex::Function` | `mexFunction` | `mexext` | `mxDuplicateArray` | `mxIsClass`

More About

- “Choosing MEX Applications” on page 7-2

MATLAB APIs

Use Matrix API and MEX API functions in gateway and computational routines to interact with data in the MATLAB workspace. These APIs are part of the MATLAB C/C++ and Fortran API Reference library.

To use these functions, include the `mex` header, which declares the entry point and interface routines. Put this statement in your source file:

```
#include "mex.h"
```

Matrix API

Use Matrix API functions to pass *mxArray*, the type MATLAB uses to store arrays, to and from MEX files. For examples using these functions, see *matlabroot/extern/examples/mx*.

MEX Library

Use MEX Library functions to perform operations in the MATLAB environment. For examples using these functions, see *matlabroot/extern/examples/mex*.

Unlike MATLAB functions, MEX file functions do not have their own variable workspace. MEX file functions operate in the caller workspace. Use `mexEvalString` to evaluate the command in the caller workspace. Use the `mexGetVariable` and `mexPutVariable` functions to get and put variables into the caller workspace.

Preprocessor Macros

The Matrix and MEX APIs use the MATLAB preprocessor macros `mwSize` and `mwIndex` for cross-platform flexibility. `mwSize` represents size values, such as array dimensions and number of elements. `mwIndex` represents index values, such as indices into arrays.

See Also

`mexEvalString` | `mexGetVariable` | `mexPutVariable` | `mwIndex` | `mwSize` | `mxArray`

More About

- “MATLAB Data” on page 7-6
- “MEX Library” on page 8-5
- C Matrix API

User Messages in C MEX Files

To print text in the MATLAB Command Window, use the `mexPrintf` function as you would a C/C++ `printf` function. To print error and warning information in the Command Window, use the `mexErrMsgIdAndTxt` and `mexWarnMsgIdAndTxt` functions in the C Matrix API.

For example, the following code snippet prints `prhs[0]`.

```
char *buf;
int buflen;

if (mxGetString(prhs[0], buf, buflen) == 0) {
    mexPrintf("The input string is: %s\n", buf);
}
```

See Also

[mexErrMsgIdAndTxt](#) | [mexPrintf](#) | [mexWarnMsgIdAndTxt](#)

Error Handling in C MEX Files

The `mexErrMsgIdAndTxt` function in the C Matrix API prints error information and terminates your MEX function. The `mexWarnMsgIdAndTxt` function prints information, but does not terminate the MEX function.

```
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
```

For information about error handling in C++ MEX functions written with the “MATLAB Data API”, see “Handling Inputs and Outputs” on page 9-37.

See Also

[mexErrMsgIdAndTxt](#) | [mexWarnMsgIdAndTxt](#)

Create C++ MEX Functions with C Matrix API

Note MATLAB provides an API that uses modern C++ semantics and design patterns, the “MATLAB Data API”. MathWorks recommends that you create MEX functions with this API. For more information, see “C++ MEX Applications”.

If your MEX functions must run in MATLAB R2017b or earlier, then you must use the “C Matrix API” functions in your C++ applications. MEX functions built with the C Matrix API support all C++ language standards. This topic discusses specific C++ language issues to consider when creating and using MEX files.

You can use the MATLAB C code examples in C++ applications. For example, see `mexcpp.cpp` in “C++ Class Example” on page 8-10, which contains both C and C++ statements.

Creating Your C++ Source File

The MATLAB C++ source code examples use the `.cpp` file extension. The extension `.cpp` is unambiguous and recognized by C++ compilers. Other possible extensions include `.C`, `.cc`, and `.cxx`.

Compiling and Linking

To build a C++ MEX file, type:

```
mex filename.cpp
```

where *filename* is the name of the source code file, on your MATLAB path.

You can run a C++ MEX file only on systems with the same version of MATLAB that the file was compiled on.

Memory Considerations for Class Destructors

Do not use the `mxFree` or `mxDestroyArray` functions in a C++ destructor of a class used in a MEX-function. If the MEX-function throws an error, MATLAB cleans up MEX-file variables, as described in “Automatic Cleanup of Temporary Arrays in MEX Files” on page 8-69.

If an error occurs that causes the object to go out of scope, MATLAB calls the C++ destructor. Freeing memory directly in the destructor means both MATLAB and the destructor free the same memory, which can corrupt memory.

Use mexPrintf to Print to MATLAB Command Window

Using `cout` or the C-language `printf` function does not work as expected in C++ MEX files. Use the `mexPrintf` function instead.

C++ Class Example

The MEX file `mexcpp.cpp` shows how to use C++ code with a C language MEX file. The example uses functions from the C Matrix API. It uses member functions, constructors, destructors, and the `iostream` include file.

The function defines a class `myData` with member functions `display` and `set_data`, and variables `v1` and `v2`. It constructs an object `d` of class `myData` and displays the initialized values of `v1` and `v2`. It then sets `v1` and `v2` to your input and displays the new values. Finally, the `delete` operator cleans up the object.

To build this example, copy the file to the MATLAB path and at the command prompt type:

```
mex mexcpp.cpp
```

The calling syntax is `mexcpp(num1, num2)`.

C++ File Handling Example

The `mexatexit.cpp` example shows C++ file handling features. Compare it with the C code example `mexatexit.c`, which uses the `mexAtExit` function.

C++ Example

The C++ example uses a `fileresource` class to handle the file open and close functions. The MEX function calls the destructor for this class, which closes the data file. This example also prints a message on the screen when performing operations on the data file. However, in this case, the only C file operation performed is the write operation, `fprintf`.

To build the `mexatexit.cpp` MEX file, copy the file to the MATLAB path and type:


```
mex mexatexit.cpp
```

Type:

```
z = 'for the C++ MEX-file';  
mexatexit(x)  
mexatexit(z)  
clear mexatexit
```

```
Writing data to file.  
Writing data to file.
```

Display the contents of `matlab.data`.

```
type matlab.data
```

```
my input string  
for the C++ MEX-file
```

C Example

The C code example registers the `mexAtExit` function to perform cleanup tasks (close the data file) when the MEX file clears. This example prints a message on the screen using `mexPrintf` when performing file operations `fopen`, `fprintf`, and `fclose`.

To build the `mexatexit.c` MEX file, copy the file to the MATLAB path and type:

```
mex mexatexit.c
```

Run the example.

```
x = 'my input string';  
mexatexit(x)
```

```
Opening file matlab.data.  
Writing data to file.
```

Clear the MEX file.

```
clear mexatexit
```

```
Closing file matlab.data.
```

Display the contents of `matlab.data`.

```
type matlab.data
```

my input string

See Also

[mexAtExit](#) | [mexPrintf](#)

Related Examples

- [mexcpp.cpp](#)
- [mexatexit.cpp](#)
- [mexatexit.c](#)

More About

- “Build C MEX Function” on page 7-16
- “C Matrix API”
- “C++ MEX Applications”
- “MATLAB Data API”

Create C Source MEX File

This example shows how to write a MEX file to call a C function `arrayProduct` in MATLAB using a MATLAB array defined in the “C Matrix API”. You can view the complete source file here on page 8-0 .

To use this example, you need:

- The ability to write C or C++ source code. You create these files with the MATLAB Editor.
- Compiler supported by MATLAB. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.
- Functions in the “C Matrix API” and the C MEX API.
- `mex` build script.

If you want to use your own C development environment, see “Custom Build with MEX Script Options” on page 7-21 for more information.

C function `arrayProduct`

The following code defines the `arrayProduct` function, which multiplies a $1 \times n$ matrix `y` by a scalar value `x` and returns the results in array `z`. You can use these same C statements in a C++ application.

```
void arrayProduct(double x, double *y, double *z, int n)
{
    int i;

    for (i=0; i<n; i++) {
        z[i] = x * y[i];
    }
}
```

Create Source File

Open MATLAB Editor, create a file, and document the MEX file with the following information.

```
/*
 * arrayProduct.c - example in MATLAB External Interfaces
 *
 * Multiplies an input scalar (multiplier)
```

```
* times a 1xN matrix (inMatrix)
* and outputs a 1xN matrix (outMatrix)
*
* The calling syntax is:
*
*     outMatrix = arrayProduct(multiplier, inMatrix)
*
* This is a MEX file for MATLAB.
*/
```

Add the C/C++ header file, `mex.h`, containing the MATLAB API function declarations.

```
#include "mex.h"
```

Save the file on your MATLAB path, for example, in `c:\work`, and name it `arrayProduct.c`. The name of your MEX file is `arrayProduct`.

Create Gateway Routine

Every C program has a `main()` function. MATLAB uses the gateway routine, `mexFunction`, as the entry point to the function. Add the following `mexFunction` code.

```
/* The gateway function */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    /* variable declarations here */

    /* code here */
}
```

This table describes the input parameters for `mexFunction`.

Parameter	Description
<code>nlhs</code>	Number of output (left-side) arguments, or the size of the <code>plhs</code> array.
<code>plhs</code>	Array of output arguments.
<code>nrhs</code>	Number of input (right-side) arguments, or the size of the <code>prhs</code> array.
<code>prhs</code>	Array of input arguments.

Verify MEX File Input and Output Parameters

Verify the number of MEX file input and output arguments using the `nrhs` and `nlhs` arguments.

To check for two input arguments, `multiplier` and `inMatrix`, use this code.

```
if(nrhs != 2) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs",
        "Two inputs required.");
}
```

Use this code to check for one output argument, the product `outMatrix`.

```
if(nlhs != 1) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs",
        "One output required.");
}
```

Verify the argument types using the `plhs` and `prhs` arguments. This code validates that `multiplier`, represented by `prhs[0]`, is a scalar.

```
/* make sure the first input argument is scalar */
if( !mxIsDouble(prhs[0]) ||
    mxIsComplex(prhs[0]) ||
    mxGetNumberOfElements(prhs[0]) != 1 ) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar",
        "Input multiplier must be a scalar.");
}
```

This code validates that `inMatrix`, represented by `prhs[1]`, is type double.

```
if( !mxIsDouble(prhs[1]) ||
    mxIsComplex(prhs[1])) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notDouble",
        "Input matrix must be type double.");
}
```

Validate that `inMatrix` is a row vector.

```
/* check that number of rows in second input argument is 1 */
if(mxGetM(prhs[1]) != 1) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notRowVector",
        "Input must be a row vector.");
}
```

Create Computational Routine

Add the `arrayProduct` code. This function is your *computational routine*, the source code that performs the functionality you want to use in MATLAB.

```
void arrayProduct(double x, double *y, double *z, int n)
{
    int i;

    for (i=0; i<n; i++) {
        z[i] = x * y[i];
    }
}
```

A computational routine is optional. Alternatively, you can place the code within the `mexFunction` function block.

Write Code for Cross-Platform Flexibility

MATLAB provides a preprocessor macro, `mwsize`, that represents size values for integers, based on the platform. The computational routine declares the size of the array as `int`. Replace the `int` declaration for variables `n` and `i` with `mwsize`.

```
void arrayProduct(double x, double *y, double *z, mwSize n)
{
    mwSize i;

    for (i=0; i<n; i++) {
        z[i] = x * y[i];
    }
}
```

Declare Variables for Computational Routine

Put the following variable declarations in `mexFunction`.

- Declare variables for the input arguments.

```
double multiplier;    /* input scalar */
double *inMatrix;    /* 1xN input matrix */
```

- Declare `ncols` for the size of the input matrix.

```
mwSize ncols;        /* size of matrix */
```

- Declare the output argument, `outMatrix`.

```
double *outMatrix;    /* output matrix */
```

Later you assign the `mexFunction` arguments to these variables.

Read Input Data

To read the scalar input, use the `mxGetScalar` function.

```
/* get the value of the scalar input */
multiplier = mxGetScalar(prhs[0]);
```

Use the `mxGetDoubles` function to point to the input matrix data.

```
/* create a pointer to the real data in the input matrix */
inMatrix = mxGetDoubles(prhs[1]);
```

Use the `mxGetN` function to get the size of the matrix.

```
/* get dimensions of the input matrix */
ncols = mxGetN(prhs[1]);
```

Prepare Output Data

To create the output argument, `plhs[0]`, use the `mxCreateDoubleMatrix` function.

```
/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(1,ncols,mxREAL);
```

Use the `mxGetDoubles` function to assign the `outMatrix` argument to `plhs[0]`

```
/* get a pointer to the real data in the output matrix */
outMatrix = mxGetDoubles(plhs[0]);
```

Perform Calculation

Pass the arguments to `arrayProduct`.

```
/* call the computational routine */
arrayProduct(multiplier,inMatrix,outMatrix,ncols);
```

View Complete Source File

Compare your source file with `arrayProduct.c` located in `matlabroot/extern/examples/mex`. Open the file `arrayProduct.c` in the editor.

For a C++ MEX file example using the “MATLAB Data API”, see `arrayProduct.cpp`. For information about creating MEX files with this API, see “C++ MEX Functions” on page 9-2.

Build MEX Function

At the MATLAB command prompt, build the function with the `mex` command.

```
mex arrayProduct.c -R2018a
```

Test the MEX Function

```
s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A)

B =
    7.5000    10.0000    45.0000
```

Validate MEX File Input Arguments

It is good practice to validate the type of a MATLAB variable before calling a MEX function. To test the input variable, `inputArg`, and convert it to `double` if necessary, use this code.

```
s = 5;
A = [1.5, 2, 9];
inputArg = int16(A);
if ~strcmp(class(inputArg), 'double')
    inputArg = double(inputArg);
end
B = arrayProduct(s,inputArg)
```

See Also

`mex` | `mexFunction` | `mwSize` | `mxCreateDoubleMatrix` | `mxGetDoubles` | `mxGetN` | `mxGetScalar`

Related Examples

- `arrayProduct.c`
- “C Matrix API”
- “Create C++ MEX Functions with C Matrix API” on page 8-9
- `arrayProduct.cpp`
- “MATLAB Data API”

- “C++ MEX Functions” on page 9-2

Tables of MEX Function Source Code Examples

The following tables contain lists of source code files for creating example MEX functions. Use these examples as a starting point for creating your own MEX functions. The tables contain the following information.

- Example Name - a link that opens the source file in MATLAB Editor for your convenience. You can use any code development editor to create source MEX files.
- Example Subfolder - the subfolder of *matlabroot/extern/examples* containing the example. Use this subfolder name when copying the file to a writable folder.
- Description - describes the example.
- More Information - a link to a topic describing or using the example, or to the API function used in the example.

Getting Started

Use the `mex` command to build the examples. Make sure that you have a MATLAB-supported compiler installed. To verify the compiler selected for the source code language *lang*, type:

```
mex -setup lang
```

Copy the file to a writable folder on your path using the following command syntax. *filename* is the name of the example, and *foldername* is the subfolder name.

```
copyfile(fullfile(matlabroot,'extern','examples','foldername','filename'),'.','f')
```

For example, to copy `arrayProduct.c`, type:

```
copyfile(fullfile(matlabroot,'extern','examples','mex','arrayProduct.c'),'.','f')
```

C, C++, and Fortran MEX Functions

To build an example MEX function in MATLAB or at your operating system prompt, use this command syntax. *filename* is the example name, and *release-option* specifies the API used by the example. For information about the MATLAB APIs, see “Choosing MEX Applications” on page 7-2.

```
mex -v -release-option filename
```

Example Name	Example Subfolder	Description	More Information
arrayFillGetPr.c	refbook	Fill mxArray using mxGetDoubles.	"Fill mxArray in C MEX File" on page 8-41
arrayFillSetData.c	refbook	Fill mxArray with non-double values.	"Fill mxArray in C MEX File" on page 8-41
arrayFillSetPr.c	refbook	Fill mxArray using mxSetDoubles to allocate memory dynamically.	"Fill mxArray in C MEX File" on page 8-41
arrayFillSetPrComplex.c	refbook	Fill mxArray using mxSetComplexDoubles to allocate memory dynamically.	"Fill mxArray in C MEX File" on page 8-41
arrayProduct.c	mex	Multiply a scalar times 1xN matrix.	"Create C Source MEX File" on page 8-13
arrayProduct.cpp	cpp_mex	Same as arrayProduct.c, using the "MATLAB Data API".	"C++ MEX Functions" on page 9-2
arraySize.c	mex	Illustrate memory requirements of large mxArray.	"Handling Large mxArrays in C MEX Files" on page 8-62
complexAdd.F	refbook	Add two complex double arrays.	
convec.c convec.F	refbook	Pass complex data.	"Handle Complex Data in C MEX File" on page 8-44
dblmat.F compute.F	refbook	Use of Fortran %VAL.	
doubleelement.c	refbook	Use unsigned 16-bit integers.	"Handle 8-, 16-, 32-, and 64-Bit Data in C MEX File" on page 8-48
explore.c	mex	Identify data type of input variable.	"Work with mxArrays" on page 4-21

Example Name	Example Subfolder	Description	More Information
findnz.c	refbook	Use N-dimensional arrays.	"Manipulate Multidimensional Numerical Arrays in C MEX Files" on page 8-49
fulltosparseIC.c fulltosparse.c fulltosparse.F, loadsparse.F	refbook	Populate a sparse matrix.	"Handle Sparse Arrays in C MEX File" on page 8-51
matsq.F	refbook	Pass matrices in Fortran.	
matsqint8.F	refbook	Pass non-double matrices in Fortran.	
mexatexit.c mexatexit.cpp	mex	Register an exit function to close a data file.	"C++ File Handling Example" on page 8-10
mexcallmatlab.c	mex	Call built-in MATLAB disp function.	
mexcallmatlabwithtrap.c	mex	How to capture error information.	
mexcpp.cpp	mex	Illustrate some C++ language features in a MEX file built with the C Matrix API.	"C++ Class Example" on page 8-10
mexevalstring.c	mex	Use mexEvalString to assign variables in MATLAB.	mexEvalString
mexfunction.c	mex	How to use mexFunction.	mexFunction

Example Name	Example Subfolder	Description	More Information
mxgetproperty.c	mex	Use <code>mxGetProperty</code> and <code>mxSetProperty</code> to change the Color property of a graphic object.	<code>mxGetProperty</code> and <code>mxSetProperty</code>
mexgetarray.c	mex	Use <code>mexGetVariable</code> and <code>mexPutVariable</code> to track counters in the MEX file and in the MATLAB global workspace.	<code>mexGetVariable</code> and <code>mexPutVariable</code>
mexgetarray.cpp		Same as <code>mexgetarray.c</code> , using <code>matlab::engine::getVariable</code> and <code>matlab::engine::setVariable</code> in the "MATLAB Data API".	"Set and Get MATLAB Variables from MEX" on page 9-57
mexlock.c mexlockf.F	mex	How to lock and unlock a MEX file.	<code>mexLock</code>
mxcalcsing lesubscript.c	mx	Demonstrate MATLAB 1-based matrix indexing versus C 0-based indexing.	<code>mxCalcSingleSubscript</code>
mxcreatecellmatrix.c mxcreatecellmatrixf.F	mx	Create 2-D cell array.	"Create 2-D Cell Array in C MEX File" on page 8-39
mxcreatecharmatrixfromstr.c	mx	Create 2-D character array.	<code>mxCreateCharMatrixFromStrings</code>
mxcreatestructarray.c	mx	Create MATLAB structure from C structure.	<code>mxCreateStructArray</code>

Example Name	Example Subfolder	Description	More Information
mxcreateuninitnumericmatrix.c	mx	Create an uninitialized mxArray, fill with local data, and return.	mxCreateUninitNumericMatrix
mxgeteps.c mxgetepsf.F	mx	Read MATLAB eps value.	mxGetEps
mxgetinf.c	mx	Read inf value.	mxGetInf
mxgetnzmax.c	mx	Display number of nonzero elements in a sparse matrix and maximum number of nonzero elements it can store.	mxGetNzmax
mxisclass.c	mx	Check if array is member of specified class.	mxIsClass
mxisfinite.c	mx	Check for NaN and infinite values.	mxIsFinite
mxislogical.c	mx	Check if workspace variable is logical or global.	mxIsLogical
mxisscalar.c	mx	Check if input variable is scalar.	mxIsScalar
mxmalloc.c	mx	Allocate memory to copy a MATLAB char array to a C-style string.	mxMalloc
mxsetdimensions.c mxsetdimensionsf.F	mx	Reshape an array.	mxSetDimensions
mxsetnzmax.c	mx	Reallocate memory for sparse matrix and reset values of pr, pi, ir, and nzmax.	mxSetNzmax
passstr.F	refbook	Pass C character matrix from Fortran to MATLAB.	

Example Name	Example Subfolder	Description	More Information
phonebook.c	refbook	Manipulate structures and cell arrays.	"Pass Structures and Cell Arrays in C MEX File" on page 8-37
phonebook.cpp	cpp_mex	Same as phonebook.c, using the "MATLAB Data API".	"C++ MEX Functions" on page 9-2
revord.c revord.F	refbook	Copy MATLAB char array to and from C-style string.	"Pass Strings in C MEX File" on page 8-32
sincall.c sincall.F, fill.F	refbook	Create mxArray and pass to MATLAB sin and plot functions.	
timestwo.c timestwo.F	refbook	Demonstrate common workflow of MEX file.	"Pass Scalar Values in C MEX File" on page 8-29
xtimesy.c xtimesy.F	refbook	Pass multiple parameters.	
yprime.c yprimef.F, yprimefg.F	mex	Solve simple three body orbit problem.	
yprime.cpp	cpp_mex	Same as yprime.c, using the "MATLAB Data API".	"C++ MEX Functions" on page 9-2

C MEX Functions Calling Fortran Subroutines

The examples in the following table call a LAPACK or BLAS function using a C MEX function. These examples link to one or both of the Fortran libraries `mwlapack` and `mwblas`. To build the MEX function, follow the instructions in the topics listed in the More Information column.

Example Name	Example Subfolder	Description	More Information
dotProductComplex.c	refbook	Handle Fortran complex return type for function called from a C MEX file.	"Handle Fortran Complex Return Type — dotProductComplex" on page 7-28
matrixDivide.c	refbook	Call a LAPACK function.	"Preserve Input Values from Modification" on page 7-23
matrixDivideComplex.c	refbook	Call a LAPACK function with complex numbers.	"Pass Complex Variables — matrixDivideComplex" on page 7-28
matrixMultiply.c	refbook	Call a BLAS function.	"Pass Arguments to Fortran Functions from C/C++ Programs" on page 7-23
utdu_slv.c	refbook	Use LAPACK for symmetric indefinite factorization.	"Symmetric Indefinite Factorization Using LAPACK — utdu_slv" on page 7-29

See Also

mex

More About

- "Choosing MEX Applications" on page 7-2
- "Build C MEX Function" on page 7-16
- "Change Default Compiler" on page 7-18
- "MATLAB Support for Interleaved Complex API in MEX Functions" on page 7-31

External Websites

- Supported and Compatible Compilers

Choose a C++ Compiler

Select Microsoft Visual Studio Compiler

This example shows how to determine and change the default compiler for building C++ MEX files when you have multiple versions of Microsoft Visual Studio® on your system. The messages in this example assume that you have Microsoft Visual C++ 2012 and Microsoft Visual C++ 2015. Use these steps for any C++ compilers on your system.

MATLAB chooses a default compiler for C source files and a default compiler for C++ source files. To see the default C++ compiler, type:

```
mex -setup c++
```

```
MEX configured to use 'Microsoft Visual C++ 2012' for C++ language compilation.
```

MATLAB also displays links to other C++ compilers installed on your system.

To change the default to MSVC 2015, click the link:

```
Microsoft Visual C++ 2015 Professional \(C\)
```

```
MEX configured to use 'Microsoft Visual C++ 2015 Professional (C)' for C++ language compilation.
```

Microsoft Visual C++ 2015 remains the default until you call `mex -setup c++` to select a different default.

When you call the `mex` command with a source file, MATLAB displays the name of the compiler used to build the MEX file. You can also find this information using the `mex -setup lang` command. By default, when you type `mex -setup`, MATLAB shows you information for the C compiler only. If you want information for C++ compilers, type:

```
mex -setup c++
```

Select MinGW-w64 Compiler

If you only have the MinGW® compiler installed on your system, the `mex` command automatically chooses MinGW for both C and C++ MEX files. If you have multiple C or C++ compilers, use `mex -setup` to choose MinGW for both C and C++ MEX files.

```
mex -setup  
mex -setup cpp
```

If you only type `mex -setup` choosing MinGW, when you compile a C++ file, `mex` might choose a different compiler.

See Also

More About

- “Change Default Compiler” on page 7-18

Pass Scalar Values in C MEX File

Pass Scalar as Matrix

This example shows how to write a MEX file that passes scalar values.

Suppose that you have the following C code, `timestwo`, that takes a scalar input, a 1-by-1 matrix, and doubles it.

```
void timestwo(double y[], double x[])
{
    y[0] = 2.0*x[0];
    return;
}
```

C Code Analysis

To see the function written as a MEX file, open `timestwo.c` in the MATLAB Editor.

In C/C++, the compiler checks function arguments for number and type. However, in MATLAB, you can pass any number or type of arguments to a function; the function is responsible for argument checking. MEX files also allow variable inputs. Your MEX file must safely handle any number of input or output arguments of any supported type.

This code checks for the proper number of arguments.

```
if(nrhs != 1) {
    mexErrMsgIdAndTxt( "MATLAB:timestwo:invalidNumInputs",
        "One input required.");
} else if(nlhs>1) {
    mexErrMsgIdAndTxt( "MATLAB:timestwo:maxlhs",
        "Too many output arguments.");
}
```

This code checks if the input is a real, scalar double value.

```
mrows = mxGetM(prhs[0]);
ncols = mxGetN(prhs[0]);
if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ||
    !(mrows==1 && ncols==1) ) {
    mexErrMsgIdAndTxt( "MATLAB:timestwo:inputNotRealScalarDouble",
```

```
    "Input must be a noncomplex scalar double.");  
}
```

Build and Test Example

Build the MEX file.

```
mex -v -R2018a timestwo.c
```

Call the function.

```
x = 2;  
y = timestwo(x)
```

```
y =  
    4
```

Pass Scalar by Value

This example shows how to write a MEX file that passes a scalar by value.

The `mxGetScalar` function returns the value of a scalar instead of a pointer to a copy of the scalar variable, `x`.

The following C code implements the `timestwo_alt` function.

```
void timestwo_alt(double *y, double x)  
{  
    *y = 2.0*x;  
}
```

Compare the `timestwo_alt` function signature with the `timestwo` function signature.

```
void timestwo_alt(double *y, double x)  
void timestwo(double y[], double x[])
```

The input value `x` is a scalar of type `double`. In the `timestwo` function, the input value is a matrix of type `double`.

To see the function written as a MEX file, open `timestwoalt.c` in the MATLAB Editor.

Compare the call to `timestwo_alt` to the call to `timestwo`.

```
/* Get the scalar value of the input x */
/* note: mxGetScalar returns a value, not a pointer */
x = mxGetScalar(prhs[0]);

/* Assign a pointer to the output */
y = mxGetDoubles(plhs[0]);

/* Call the timestwo_alt subroutine */
timestwo_alt(y,x);

/* Assign pointers to each input and output. */
x = mxGetDoubles(prhs[0]);
y = mxGetDoubles(plhs[0]);

/* Call the timestwo subroutine. */
timestwo(y,x);
```

The value `x` created by `mxGetScalar` is a scalar not a pointer.

Pass Strings in C MEX File

This example shows how to pass strings to a MEX function built with the C Matrix API. The example `revord.c` accepts a character vector and returns the characters in reverse order.

C Code Analysis

To see the code, open `revord.c` in the MATLAB Editor.

The gateway function, `mexFunction`, creates a C string from the input variable, `prhs[0]`. By isolating variables of type `mxArray` from the computational subroutine, `revord`, you can avoid making significant changes to your original C and C++ code.

Convert the input argument `prhs[0]` to a C-style string `input_buf`.

```
input_buf = mxArrayToString(prhs[0]);
```

Allocate memory for the output argument, `output_buf`, a C-style string.

```
output_buf = mxMalloc(buflen, sizeof(char));
```

The size of the output argument is equivalent to the size of the input argument.

Call the computational subroutine, `revord`.

```
revord(input_buf, buflen, output_buf);
```

Convert the output, `output_buf`, to an `mxArray` and assign to `plhs[0]`.

```
plhs[0] = mxCreateString(output_buf);
```

Do not release memory for this variable because it is an output argument.

The `mxArrayToString` function, used to create the temporary `input_buf` variable, allocates memory; use the `mxFree` function to release the memory.

```
mxFree(input_buf);
```

Build and Test Example

Run the following commands from the MATLAB command line.

Build the example.

```
mex -v revord.c
```

Call the function.

```
x = 'hello world';  
y = revord(x)
```

```
y =  
dlrow olleh
```

See Also

Related Examples

- `revord.c`

C Matrix API String Handling Functions

In this section...
“How MATLAB Represents Strings in MEX Files” on page 8-34
“Character Encoding and Multibyte Encoding Schemes” on page 8-34
“Converting MATLAB Character Vector to C-Style String” on page 8-35
“Converting C-Style String to MATLAB Character Vector” on page 8-35
“Returning Modified Input String” on page 8-35
“Memory Management” on page 8-35

How MATLAB Represents Strings in MEX Files

In C/C++ MEX functions built with the C Matrix API, a MATLAB character vector is an `mxArray` of type `mxChar`, using a locale-neutral data representation (Unicode encoding). MATLAB represents C-style strings as type `char`, and uses the character encoding scheme specified by the user locale setting.

The following C Matrix API functions provide string handling functions to help you work with both `mxArrays` and C-style strings.

- `mxCreateString` — Creates a `mxChar` `mxArray` initialized to the input string.
- `mxArrayToString` — Copies a `mxChar` `mxArray` into a C-style string. Supports multibyte encoded characters.
- `mxGetString` — Copies a `mxChar` `mxArray` into a C-style string. Best used with single-byte encoded characters. Supports multibyte encoded characters when you calculate string buffer size.
- `mxGetChars` — Returns a pointer to the first `mxChar` element in the `mxArray`.

Consider the following topics when choosing a string handling function.

Character Encoding and Multibyte Encoding Schemes

MATLAB supports the character encoding scheme specified by the user locale setting. When an MX Library function converts `mxChar` data to a C `char` type, MATLAB also converts the character to the user default encoding.

If you use a multibyte encoding scheme, use the `mxArrayToString` function.

The `mxGetChars` function provides a pointer to the `mxChar` array; it does not change the character encoding.

You can also use the `mxGetString` function with multibyte encoding schemes. `mxGetString` converts the `mxChar` data to your user default encoding, and copies the converted characters to the destination buffer. However, you must calculate the size of the destination buffer. For single-byte encoding, the size of the buffer is the number of characters, plus 1 for the null terminator. For multibyte encoding, the size of a character is one or more bytes. Some options for calculating the buffer size are to overestimate the amount (calculating the number of characters times the maximum number of bytes used by the encoding scheme), analyze the string to determine the precise size used by each character, or utilize 3rd-party string buffer libraries. After this calculation, add 1 for the null terminator.

Converting MATLAB Character Vector to C-Style String

When you pass a character array to a MEX function, it is an `mxArray` of type `mxChar`. If you call a C function to manipulate the string, first convert the data to a C type `char` using the `mxArrayToString` or `mxGetString` functions.

Converting C-Style String to MATLAB Character Vector

If your MEX file creates a C string and returns the data to MATLAB, use the `mxCreateString` function to copy the C string into an `mxChar` array.

Returning Modified Input String

Suppose that your MEX file takes character input, modifies it, and returns the result. Since MEX file input parameters (the `prhs` array) are read-only, you must define a separate output parameter to handle the modified string.

Memory Management

MathWorks recommends that MEX file functions destroy their own temporary arrays and free their own dynamically allocated memory. The function you use to release memory depends on how you use the string buffer and what function you use to create the buffer.

If You Call This Function	Release Memory Using This Function
Any string function listed here	Do not destroy an mxArray in a source MEX file when it is: <ul style="list-style-type: none"> • Passed to the MEX file in the right-hand side list prhs []. • Returned in the left side list plhs []. • Returned by the mexGetVariablePtr function. • Used to create a structure.
mxArrayToString	mxFree
mxGetString	When using mxMalloc / mxMalloc / mxRealloc to create input argument buf, call mxFree(buf).
mxCreateString	mxDestroyArray
mxGetChars	None. Function creates a pointer to an mxArray but does not allocate additional memory.

See Also

More About

- “C Matrix API”
- “Locale Settings for MATLAB Process”

Pass Structures and Cell Arrays in C MEX File

Passing structures and cell arrays into MEX files is like passing any other data type, except the data itself in the C Matrix API is of type `mxArray`. In practice, `mxGetField` (for structures) and `mxGetCell` (for cell arrays) return pointers of type `mxArray`. You treat the pointers like any other pointers of type `mxArray`. To pass the data contained in the `mxArray` to a C routine, use an API function such as `mxGetData` to access it.

This MEX file example uses the C Matrix API. For a C++ MEX file example using the “MATLAB Data API”, see `phonebook.cpp`. For information about creating MEX files with this API, see “C++ MEX Functions” on page 9-2.

This example takes an `m-by-n` structure matrix as input and returns a new `1-by-1` structure that contains these fields:

- Text input generates an `m-by-n` cell array
- Numeric input (noncomplex, scalar values) generates an `m-by-n` vector of numbers with the same class ID as the input, for example `int`, `double`, and so on.

To build this example, at the command prompt type:

```
mex phonebook.c
```

To see how this program works, create this structure:

```
friends(1).name = 'Jordan Robert';  
friends(1).phone = 3386;  
friends(2).name = 'Mary Smith';  
friends(2).phone = 3912;  
friends(3).name = 'Stacy Flora';  
friends(3).phone = 3238;  
friends(4).name = 'Harry Alpert';  
friends(4).phone = 3077;
```

Call the MEX file:

```
phonebook(friends)
```

```
ans =  
    name: {1x4 cell }  
    phone: [3386 3912 3238 3077]
```

See Also

Related Examples

- `phonebook.c`
- “Tables of MEX Function Source Code Examples” on page 8-20
- `phonebook.cpp`
- “C++ MEX Functions” on page 9-2

Create 2-D Cell Array in C MEX File

This example shows how to create a cell array in a MEX function, using the `mxcreatecellmatrix.c` function, which places input arguments in a cell array.

C Code Analysis

To see the code, open `mxcreatecellmatrix.c` in the MATLAB Editor.

Create a cell array for the number of input arguments.

```
cell_array_ptr = mxCreateCellMatrix((mwSize)nrhs,1);
```

Copy the input arguments into the cell array.

```
for( i=0; i<(mwIndex)nrhs; i++){  
    mxSetCell(cell_array_ptr,i,mxDuplicateArray(prhs[i]));
```

Build and Test Example

Run the following commands from the MATLAB command line.

Build the example.

```
mex -v mxcreatecellmatrix.c
```

Create input arguments.

```
str1 = 'hello';  
str2 = 'world';  
num = 2012;
```

Create a 3-x-1 cell array and call `disp` to display the contents.

```
mxcreatecellmatrix(str1,str2,num)
```

The contents of the created cell is:

```
'hello'
```

```
'world'  
[2012]
```

See Also

Related Examples

- `mxcreatecellmatrix.c`
- “Tables of MEX Function Source Code Examples” on page 8-20

Fill mxArray in C MEX File

Options

You can move data from a C MEX file into an mxArray using the C Matrix API. The functions you use depend on the type of data in your application. Use the `mxSetDoubles` and `mxGetDoubles` functions for data of type `double`. For numeric data other than `double`, use the one of the typed data access functions. For nonnumeric data, see the examples for the `mxCreateString` function.

The following examples use a variable `data` to represent data from a computational routine. Each example creates an mxArray using the `mxCreateNumericMatrix` function, fills it with `data`, and returns it as the output argument `plhs[0]`.

If you have complex data or the type is not `double`, then use the “Typed Data Access in C MEX Files” on page 8-65 functions. The typed data access functions are part of the interleaved complex C Matrix API; use the `mex -R2018a` option to build the MEX functions.

Copying Data Directly into an mxArray

The `arrayFillGetPr.c` example uses the `mxGetDoubles` function to copy the values from `data` to `plhs[0]`.

Pointing to Data

The `arrayFillSetPr.c` example uses the `mxSetDoubles` function to point `plhs[0]` to `data`. The `arrayFillSetPrComplex.c` example uses the `mxSetComplexDoubles` function to point to complex data.

The example `arrayFillSetData.c` shows how to fill an mxArray for numeric types other than `double`.

See Also

`mxCreateString` | `mxGetDoubles` | `mxSetComplexDoubles` | `mxSetDoubles`

Related Examples

- `arrayFillGetPr.c`
- `arrayFillSetPr.c`
- `arrayFillSetData.c`
- `arrayFillSetPrComplex.c`
- “Tables of MEX Function Source Code Examples” on page 8-20

Prompt User for Input in C MEX File

Because MATLAB does not use `stdin` and `stdout`, do not use C/C++ functions like `scanf` and `printf` to prompt for user input. The following example shows how to use `mexCallMATLAB` with the `input` function to get a number from the user.

```
#include "mex.h"
#include "string.h"
void mexFunction( int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[] )
{
    mxArray  *new_number, *str;
    double out;

    str = mxCreateString("Enter extension: ");
    mexCallMATLAB(1,&new_number,1,&str,"input");
    out = mxGetScalar(new_number);
    mexPrintf("You entered: %.0f ", out);
    mxDestroyArray(new_number);
    mxDestroyArray(str);
    return;
}
```

See Also

[input](#) | [inputdlg](#) | [mexCallMATLAB](#)

Related Examples

- “Tables of MEX Function Source Code Examples” on page 8-20

Handle Complex Data in C MEX File

The example `convec.c` takes two complex row vectors and convolves them. The MEX file uses functions in the C Matrix API.

Create Source File

The following statements document the MEX function.

```
/*=====
 * convec.c
 * example for passing complex data from MATLAB to C and back again
 *
 * convolves two complex input vectors
 *
 * This is a MEX-file for MATLAB.
 * Copyright 1984-2017 The MathWorks, Inc.
 *=====*/
```

Create Gateway Routine and Verify Input and Output Parameters

The following statements add the C/C++ header file `mex.h` and creates the `mexFunction` entry point.

```
#include "mex.h"

/* The gateway routine. */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    return;
}
```

The following statements validate the number of parameters.

```
/* check for the proper number of arguments */
if(nrhs != 2)
    mexErrMsgIdAndTxt( "MATLAB:convec:invalidNumInputs",
                      "Two inputs required.");
if(nlhs > 1)
```

```
mexErrMsgIdAndTxt( "MATLAB:convec:maxlhs",
    "Too many output arguments.");
```

The following statement verifies that the input arguments are row vectors.

```
/*Check that both inputs are row vectors*/
if( mxGetM(prhs[0]) != 1 || mxGetM(prhs[1]) != 1 )
    mexErrMsgIdAndTxt( "MATLAB:convec:inputsNotVectors",
        "Both inputs must be row vectors.");
```

The following statement verifies that the input arguments are complex.

```
/* Check that both inputs are complex*/
if( !mxIsComplex(prhs[0]) || !mxIsComplex(prhs[1]) )
    mexErrMsgIdAndTxt( "MATLAB:convec:inputsNotComplex",
        "Inputs must be complex.\n");
```

Create Output mxArray

The following statements define the parameters for creating the output mxArray.

```
size_t rows, cols;
size_t nx, ny;

/* get the length of each input vector */
nx = mxGetN(prhs[0]);
ny = mxGetN(prhs[1]);

rows = 1;
cols = nx + ny - 1;
```

The following statement creates an array and sets the output pointer `plhs[0]` to it.

```
plhs[0] = mxCreateDoubleMatrix( (mwSize)rows, (mwSize)cols, mxCOMPLEX);
```

Create Computational Routine

The following statements define `convec`, the computational routine.

```
void convec(mxArray * x, mxArray * y, mxArray * z,
    size_t nx, size_t ny)
{
    mwSize i,j;

    /* get pointers to the complex arrays */
    mxComplexDouble * xc = mxGetComplexDoubles(x);
```

```
mxComplexDouble * yc = mxGetComplexDoubles(y);
mxComplexDouble * zc = mxGetComplexDoubles(z);
zc[0].real = 0;
zc[0].imag = 0;
/* perform the convolution of the complex vectors */
for(i=0; i<nx; i++) {
    for(j=0; j<ny; j++) {
        zc[i+j].real =
            zc[i+j].real + xc[i].real * yc[j].real - xc[i].imag * yc[j].imag;

        zc[i+j].imag =
            zc[i+j].imag + xc[i].real * yc[j].imag + xc[i].imag * yc[j].real;
    }
}
}
```

Call convec

The following statement calls the computational routine.

```
convec(prhs[0], prhs[1], plhs[0], nx, ny);
```

Build and Test

At the MATLAB command prompt type:

```
mex -R2018a convec.c
```

Test the MEX file.

```
x = [3.000 - 1.000i, 4.000 + 2.000i, 7.000 - 3.000i];
y = [8.000 - 6.000i, 12.000 + 16.000i, 40.000 - 42.000i];
z = convec(x,y)
```

```
z =
    1.0e+02 *
```

```
Columns 1 through 4
```

```
0.1800 - 0.2600i 0.9600 + 0.2800i 1.3200 - 1.4400i 3.7600 - 0.1200i
```

```
Column 5
```

```
1.5400 - 4.1400i
```

Compare the results with the built-in MATLAB function `conv`.

```
conv(x,y)
```

See Also

`mxGetComplexDoubles`

Related Examples

- `convec.c`
- “Tables of MEX Function Source Code Examples” on page 8-20

Handle 8-, 16-, 32-, and 64-Bit Data in C MEX File

The C Matrix API provides a set of functions that support signed and unsigned 8-, 16-, 32-, and 64-bit data. For example, the `mxCreateNumericArray` function constructs an unpopulated N-dimensional numeric array with a specified data size. For more information, see `mxClassID`.

Once you have created an unpopulated MATLAB array of a specified data type, you can access the data using the typed data access functions, like `mxGetInt8s` and `mxGetComplexInt8s`. You can perform arithmetic on data of 8-, 16-, 32-, or 64-bit precision in MEX files. MATLAB recognizes the correct data class of the result.

The example `doubleelement.c` constructs a 2-by-2 matrix with unsigned 16-bit integers, doubles each element, and returns both matrices to MATLAB.

To build this example, at the command prompt type:

```
mex -R2017b doubleelement.c
```

Call the example.

```
doubleelement
```

```
ans =  
    2     6  
    4     8
```

The output of this function is a 2-by-2 matrix populated with unsigned 16-bit integers.

See Also

`mxClassID` | `mxCreateNumericArray` | `mxGetComplexInt8s` | `mxGetInt8s`

Related Examples

- `doubleelement.c`
- “Tables of MEX Function Source Code Examples” on page 8-20

Manipulate Multidimensional Numerical Arrays in C MEX Files

You can manipulate multidimensional numerical arrays by using typed data access functions like `mxGetDoubles` and `mxGetComplexDoubles`. The example `findnz.c` takes an N-dimensional array of doubles and returns the indices for the nonzero elements in the array.

Build the example.

```
mex -R2018a findnz.c
```

Create a sample matrix.

```
matrix = [ 3 0 9 0; 0 8 2 4; 0 9 2 4; 3 0 9 3; 9 9 2 0]
```

```
matrix =  
  3   0   9   0  
  0   8   2   4  
  0   9   2   4  
  3   0   9   3  
  9   9   2   0
```

`findnz` determines the position of all nonzero elements in the matrix.

```
nz = findnz(matrix)
```

```
nz =  
  1   1  
  4   1  
  5   1  
  2   2  
  3   2  
  5   2  
  1   3  
  2   3  
  3   3  
  4   3  
  5   3  
  2   4  
  3   4  
  4   4
```

See Also

`mxGetComplexDoubles` | `mxGetDoubles`

Related Examples

- `findnz.c`
- “Tables of MEX Function Source Code Examples” on page 8-20

Handle Sparse Arrays in C MEX File

The C Matrix API provides a set of functions that allow you to create and manipulate sparse arrays from within your MEX files. These API routines access and manipulate `ir` and `jc`, two of the parameters associated with sparse arrays. For more information on how MATLAB stores sparse arrays, see “The MATLAB Array” on page 7-6.

The example `fulltosparseIC.c` shows how to populate a sparse matrix.

Build the example.

```
mex -R2018a fulltosparseIC.c
```

Create a full, 5-by-5 identity matrix.

```
full = eye(5)
```

```
full =  
    1    0    0    0    0  
    0    1    0    0    0  
    0    0    1    0    0  
    0    0    0    1    0  
    0    0    0    0    1
```

Call `fulltosparse` to produce the corresponding sparse matrix.

```
spar = fulltosparse(full)
```

```
spar =  
    (1,1)    1  
    (2,2)    1  
    (3,3)    1  
    (4,4)    1  
    (5,5)    1
```

See Also

Related Examples

- `fulltosparseIC.c`
- `fulltosparse.c`

- “Tables of MEX Function Source Code Examples” on page 8-20

Debug on Microsoft Windows Platforms

This example shows how to debug `yprime.c`, found in your `matlabroot/extern/examples/mex/` folder, with Microsoft Visual Studio 2012. If you are using a different version, refer to the corresponding Microsoft documentation.

- 1 Make sure Visual Studio is your selected C compiler:

```
cc = mex.getCompilerConfigurations('C','Selected');  
cc.Name
```

```
ans =
```

```
Microsoft Visual C++ 2012 (C)
```

- 2 Compile the source MEX file with the `-g` option, which builds the file with debugging symbols included. For example:

```
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'),'.','f')  
mex -g yprime.c
```

- 3 Start Visual Studio. Do not exit your MATLAB session.
- 4 From the Visual Studio **Tools** menu, select **Attach to Process**
- 5 In the Attach to Process dialog box, select the MATLAB process, and click **Attach**.

Visual Studio loads data then displays an empty code pane.

- 6 Open the source file `yprime.c` by selecting **File > Open > File**. Locate `yprime.c` in the folder, `c:\work`.
- 7 Set a breakpoint by right-clicking the desired line of code and following **Breakpoint > Insert Breakpoint** on the context menu. It is often convenient to set a breakpoint at `mexFunction` to stop at the beginning of the gateway routine.

If you have not yet run the executable file, ignore any “!” icon that appears with the breakpoint next to the line of code.

Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

- 8 Open MATLAB and type:

```
yprime(1,1:4)
```

`yprime.c` is opened in the Visual Studio debugger at the first breakpoint.

- 9 If you select **Debug > Continue**, MATLAB displays:

```
ans =  
    2.0000    8.9685    4.0000   -1.0947
```

For more information on how to debug in the Visual Studio environment, see your Microsoft documentation.

Notes on Debugging

Binary MEX files built with the `-g` option do not execute on other computers because they rely on files that are not distributed with MATLAB. For more information on isolating problems with MEX files, see Troubleshooting on “C MEX File Applications”.

Do not enable "Access Violation" for handling exceptions when debugging MEX files. To avoid breaking at this exception, clear the check box for **Win32 Exceptions**. For Visual Studio 2015, look for the check box in **Debug > Windows > Exception Settings...** If you are using a different version of Visual Studio, refer to the corresponding Microsoft documentation.

See Also

More About

- “Debug in Simulink Environment” (Simulink)
- “MATLAB Code Analysis” (MATLAB Coder)

External Websites

- How to debug MEX-file compiled with MinGW64 and `-g` flags

Debug on Linux Platforms

The GNU® Debugger `gdb`, available on Linux systems, provides complete source code debugging, including the ability to set breakpoints, examine variables, and step through the source code line-by-line.

In this procedure, the MATLAB command prompt `>>` is shown in front of MATLAB commands, and `linux>` represents a Linux prompt; your system might show a different prompt. The debugger prompt is `<gdb>`.

To debug with `gdb`:

- 1 Compile the source MEX file with the `-g` option, which builds the file with debugging symbols included. For this example, at the Linux prompt, type:

```
linux> mex -g yprime.c
```

- 2 At the Linux prompt, start the `gdb` debugger using the `matlab -D` option.

```
linux> matlab -Dgdb
```

- 3 Tell `gdb` to stop for debugging.

```
<gdb> handle SIGSEGV SIGBUS nostop noprint  
<gdb> handle SIGUSR1 stop print
```

- 4 Start MATLAB without the Java Virtual Machine (JVM) by using the `-nojvm` startup flag.

```
<gdb> run -nojvm
```

- 5 In MATLAB, enable debugging with the `dbmex` function and run your binary MEX file.

```
>> dbmex on  
>> yprime(1,1:4)
```

- 6 You are ready to start debugging.

It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

```
<gdb> break mexFunction  
<gdb> r
```

- 7 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type:

```
<gdb> continue
```

- 8** After stopping at the last breakpoint, type:

```
<gdb> continue
```

yprime finishes and MATLAB displays:

```
ans =
```

```
    2.0000    8.9685    4.0000   -1.0947
```

- 9** From the MATLAB prompt you can return control to the debugger by typing:

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type:

```
>> quit
```

- 10** When you are finished with the debugger, type:

```
<gdb> quit
```

You return to the Linux prompt.

Refer to the documentation provided with your debugger for more information on its use.

See Also

dbmex

Debug on Mac Platforms

Using Xcode

This example shows how to debug the `yprime` MEX file using Xcode.

Copy Source MEX File

The `yprime.c` source code is in the `matlabroot` folder. In MATLAB, copy the file to a local, writable folder, for example `/Users/Shared/work`. Create the folder if it does not exist, and set it as your current folder in MATLAB.

```
workdir = fullfile('/', 'Users', 'Shared', 'work');
mkdir(workdir)
copyfile(fullfile(matlabroot, 'extern', 'examples', 'mex', 'yprime.c'), workdir)
cd(workdir)
```

Compile Source MEX File

Compile the source MEX file with the `-g` option, which adds debugging symbols. MATLAB creates the binary MEX file, `yprime.mexmaci64`.

```
mex -g yprime.c
```

Create Empty Xcode Workspace for Debugging

In Xcode,

- Select **File > New > Workspace**.
- In the file selection dialog box, set the Workspace name in the **Save As:** field to `debug_yprime`.
- Select the `/Users/Shared/work` folder in which to store the workspace. To select the folder, either navigate to the folder, or press the **Command+Shift+G** keyboard shortcut to toggle the **Go to the folder:** menu and type the full path `/Users/Shared/work`.
- Click **Save**.

Add `yprime` Files to Xcode Workspace

- To add the `yprime.c` file to the workspace, drag it from the `/Users/Shared/work` folder in the Finder into the navigator column on the left side of the Xcode workspace window.

- Clear the **Destination** option, Copy items into destination group's folder (if needed). Clearing this option enables breakpoints to be added to the file that MATLAB runs.
- To add the file, click **Finish**.

Create Scheme

- Select **Product > Scheme > New Scheme...**
- Leave **Target** set to None.
- Set **Name** to debug.
- Press **OK**. The scheme editing dialog box opens.
- Set the **Run > Info > Executable** option to **Other...** In the file selection window, press the **Command+Shift+G** keyboard shortcut to toggle the **Go to the folder:** menu. Specify the full path to the MATLAB_maci64 executable inside the MATLAB application bundle. An example of a full path is `/Applications/MATLAB_R2016a.app/Contents/MacOS/MATLAB_maci64`.
- Select **Wait for executable to be launched**.
- Click **Close**.

Add Symbolic Breakpoint

- Select **Debug > Breakpoints > Create Symbolic Breakpoint**.
- Set **Symbol** to `NSApplicationMain`.
- To add the following debugger command, click **Add action**:

```
process handle -p true -n false -s false SIGSEGV SIGBUS
```
- If the breakpoint editor pane disappears, right-click the new breakpoint and select **Edit Breakpoint...** to get back to it
- Check **Automatically continue after evaluating actions**.

Set Breakpoints in Your MEX File

- Select **View > Navigators > Show Project Navigator**.
- Click `yprime.c` in the navigator column.
- Click the gutter next to the line where you want execution to pause, for example, at the first line in `mexFunction()`.
- For more information, refer to the Xcode documentation.

Start Xcode Debugger and Run MATLAB

- To start the debugger, in Xcode select **Product > Run**. Alternatively, click the **Run** button with the triangle icon near the top left corner of the workspace window.
- Wait for Xcode to display the message **Waiting for MATLAB to launch** at the top of the Workspace window. This action might take some seconds, especially the first time you use this procedure.
- Start the MATLAB executable from the Mac Terminal prompt (see “Start from Terminal Window”) or from the Finder. If MATLAB is already running, right-click the MATLAB icon in the Dock and select **Open Additional Instance of MATLAB**.
- Xcode displays the message **Running MATLAB: debug**.

Run Binary MEX File in MATLAB

In this new instance of MATLAB, change the current folder to the folder with the `yprime` files and run the MEX file.

```
workdir = fullfile('/', 'Users', 'Shared', 'work');  
cd(workdir)  
yprime(1,1:4)
```

The Xcode debugger halts in `yprime.c` at the first breakpoint.

At this point you can step through your code, examine variables, etc., but for this exercise, select **Continue** from the **Debug** menu. The execution of `yprime` finishes and MATLAB displays:

```
ans =  
  
    2.0000    8.9685    4.0000   -1.0947
```

As long as this instance of MATLAB continues running, you can execute your MEX file repeatedly and Xcode halts at the breakpoints you set.

Using LLDB

LLDB is the debugger available with Xcode on macOS systems. Refer to the documentation provided with your debugger for more information on its use.

In this procedure, `>>` indicates the MATLAB command prompt, and `%` represents a Mac Terminal prompt. The debugger prompt is `(lldb)`.

Debug MEX Without JVM

This example debugs the `yprime` MEX file without the Java Virtual Machine (JVM). Running MATLAB in this mode minimizes memory usage and improves initial startup speed, but restricts functionality. For example, you cannot use the desktop.

- 1 Compile the source MEX file with the `-g` option, which builds the file with debugging symbols included. At the Terminal prompt, type:

```
% mex -g yprime.c
```

- 2 Start the `lldb` debugger using the `matlab` function `-D` option:

```
% matlab -Dlldb
```

- 3 Start MATLAB using the `-nojvm` startup flag:

```
(lldb) run -nojvm
```

- 4 In MATLAB, enable debugging with the `dbmex` function and run your MEX file:

```
>> dbmex on  
>> yprime(1,1:4)
```

The debugger traps a user-defined signal and the prompt returns to `lldb`.

- 5 You are ready to start debugging.

It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

```
(lldb) b mexFunction
```

- 6 Once you hit a breakpoint, you can use any debugger commands to examine variables, display memory, or inspect registers. To proceed from a breakpoint, type:

```
(lldb) c
```

- 7 After stopping at the last breakpoint, type:

```
(lldb) c
```

`yprime` finishes and MATLAB displays:

```
ans =  
    2.0000    8.9685    4.0000   -1.0947
```

- 8** From the MATLAB prompt, return control to the debugger by typing:

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type:

```
>> quit
```

- 9** When you are finished with the debugger, type:

```
(lldb) q
```

You return to the Terminal prompt.

Debug MEX with JVM

To debug a MEX file with the JVM, first handle SIGSEGV and SIGBUS process signals. Start MATLAB and stop at the first instruction.

- At the Terminal prompt, compile the MEX file and start the lldb debugger.

```
% mex -g yprime.c  
% matlab -Dlldb
```

- Start MATLAB.

```
(lldb) process launch -s
```

- Tell the process to continue when these process signals occur.

```
(lldb) process handle -p true -n false -s false SIGSEGV SIGBUS
```

- You can set break points and execute other debugger commands.

See Also

More About

- “Start MATLAB on macOS Platforms”

Handling Large mxArray in C MEX Files

Binary MEX files built on 64-bit platforms can handle 64-bit mxArray. These large data arrays can have up to $2^{48}-1$ elements. The maximum number of elements a sparse mxArray can have is $2^{48}-2$.

Using the following instructions creates platform-independent binary MEX files as well.

Your system configuration can affect the performance of MATLAB. The 64-bit processor requirement enables you to create the mxArray and access data in it. However, the system memory, in particular the size of RAM and virtual memory, determine the speed at which MATLAB processes the mxArray. The more memory available, the faster the processing.

The amount of RAM also limits the amount of data you can process at one time in MATLAB. For guidance on memory issues, see “Strategies for Efficient Use of Memory”.

Using the 64-Bit API

The signatures of the API functions shown in the following table use the `mwSize` or `mwIndex` types to work with a 64-bit mxArray. The variables you use in your source code to call these functions must be the correct type.

C mxArray Functions Using mwSize/mwIndex

mxCalcSingleSubscript	mxGetJc
mxCalloc	mxGetM
mxCreateCellArray	mxGetN
mxCreateCellMatrix	mxGetNumberOfDimensions
mxCreateCharArray	mxGetNumberOfElements
mxCreateCharMatrixFromStrings	mxGetNzmax
mxCreateDoubleMatrix	mxGetProperty
mxCreateLogicalArray	mxGetString
mxCreateLogicalMatrix	mxMalloc
mxCreateNumericArray	mxRealloc
mxCreateNumericMatrix	mxSetCell
mxCreateSparse	mxSetDimensions
mxCreateSparseLogicalMatrix	mxSetField
mxCreateStructArray	mxSetFieldByNumber
mxCreateStructMatrix	mxSetIr
mxGetCell	mxSetJc
mxGetDimensions	mxSetM
mxGetElementSize	mxSetN
mxGetField	mxSetNzmax
mxGetFieldByNumber	mxSetProperty
mxGetIr	

Example

The example, `arraySize.c` in `matlabroot/extern/examples/mex`, shows memory requirements of large mxArray. To see the example, open `arraySize.c` in the MATLAB Editor.

This function requires one positive scalar numeric input, which it uses to create a square matrix. It checks the size of the input to make sure that your system can theoretically create a matrix of this size. If the input is valid, it displays the size of the mxArray in kilobytes.

Build this MEX file.

```
mex arraySize.c
```

Run the MEX file.

```
arraySize(2^10)
```

```
Dimensions: 1024 x 1024
```

```
Size of array in kilobytes: 1024
```

If your system does not have enough memory to create the array, MATLAB displays an Out of memory error.

You can experiment with this function to test the performance and limits of handling large arrays on your system.

Caution Using Negative Values

When using the 64-bit API, `mwSize` and `mwIndex` are equivalent to `size_t` in C/C++. This type is unsigned, unlike `int`, which is the type used in the 32-bit API. Be careful not to pass any negative values to functions that take `mwSize` or `mwIndex` arguments. Do not cast negative `int` values to `mwSize` or `mwIndex`; the returned value cannot be predicted. Instead, change your code to avoid using negative values.

Building Cross-Platform Applications

If you develop programs that can run on both 32-bit and 64-bit architectures, pay attention to the upper limit of values for `mwSize` and `mwIndex`. The 32-bit application reads these values and assigns them to variables declared as `int` in C/C++. Be careful to avoid assigning a large `mwSize` or `mwIndex` value to an `int` or other variable that might be too small.

See Also

`mex` | `mwIndex` | `mwSize`

Typed Data Access in C MEX Files

The functions `mxGetPr` and `mxGetPi` in the C and Fortran Matrix APIs read data elements in `mxArrays` of type `mxDOUBLE_CLASS`. However, these functions do not verify the array type of the input argument. For type-safe data access, use the C `mxGetDoubles` and `mxGetComplexDoubles` functions or the Fortran `mxGetDoubles` and `mxGetComplexDoubles` functions. There are typed data access functions for each numeric `mxArray` type, as shown in this table.

The typed data access functions are part of the interleaved complex C and Fortran Matrix APIs; use the `mex -R2018a` option to build the MEX functions.

MATLAB mxArray Types	C Typed Data Access Functions	Fortran Typed Data Access Functions
<code>mxDOUBLE_CLASS</code>	<code>mxGetDoubles</code> <code>mxSetDoubles</code> <code>mxGetComplexDoubles</code> <code>mxSetComplexDoubles</code>	<code>mxGetDoubles</code> <code>mxSetDoubles</code> <code>mxGetComplexDoubles</code> <code>mxSetComplexDoubles</code>
<code>mxSINGLE_CLASS</code>	<code>mxGetSingles</code> <code>mxSetSingles</code> <code>mxGetComplexSingles</code> <code>mxSetComplexSingles</code>	<code>mxGetSingles</code> <code>mxSetSingles</code> <code>mxGetComplexSingles</code> <code>mxSetComplexSingles</code>
<code>mxINT8_CLASS</code>	<code>mxGetInt8s</code> <code>mxSetInt8s</code> <code>mxGetComplexInt8s</code> <code>mxSetComplexInt8s</code>	<code>mxGetInt8s</code> <code>mxSetInt8s</code> <code>mxGetComplexInt8s</code> <code>mxSetComplexInt8s</code>
<code>mxUINT8_CLASS</code>	<code>mxGetUint8s</code> <code>mxSetUint8s</code> <code>mxGetComplexUint8s</code> <code>mxSetComplexUint8s</code>	<code>mxGetUint8s</code> <code>mxSetUint8s</code> <code>mxGetComplexUint8s</code> <code>mxSetComplexUint8s</code>
<code>mxINT16_CLASS</code>	<code>mxGetInt16s</code> <code>mxSetInt16s</code> <code>mxGetComplexInt16s</code> <code>mxSetComplexInt16s</code>	<code>mxGetInt16s</code> <code>mxSetInt16s</code> <code>mxGetComplexInt16s</code> <code>mxSetComplexInt16s</code>

MATLAB mxArray Types	C Typed Data Access Functions	Fortran Typed Data Access Functions
mxUINT16_CLASS	mxGetUint16s mxSetUint16s mxGetComplexUint16s mxSetComplexUint16s	mxGetUint16s mxSetUint16s mxGetComplexUint16s mxSetComplexUint16s
mxINT32_CLASS	mxGetInt32s mxSetInt32s mxGetComplexInt32s mxSetComplexInt32s	mxGetInt32s mxSetInt32s mxGetComplexInt32s mxSetComplexInt32s
mxUINT32_CLASS	mxGetUint32s mxSetUint32s mxGetComplexUint32s mxSetComplexUint32s	mxGetUint32s mxSetUint32s mxGetComplexUint32s mxSetComplexUint32s
mxINT64_CLASS	mxGetInt64s mxSetInt64s mxGetComplexInt64s mxSetComplexInt64s	mxGetInt64s mxSetInt64s mxGetComplexInt64s mxSetComplexInt64s
mxUINT64_CLASS	mxGetUint64s mxSetUint64s mxGetComplexUint64s mxSetComplexUint64s	mxGetUint64s mxSetUint64s mxGetComplexUint64s mxSetComplexUint64s

See Also

More About

- `explore.c`
- `complexAdd.F`
- “Tables of MEX Function Source Code Examples” on page 8-20

Persistent mxArray's

You can exempt an array, or a piece of memory, from the MATLAB automatic cleanup by calling `mexMakeArrayPersistent` or `mexMakeMemoryPersistent`. However, if a MEX function creates persistent objects, then a memory leak could occur if the MEX function is cleared before the persistent object is properly destroyed. To prevent memory leaks, use the `mexAtExit` function to register a function to free the memory for objects created using these functions.

The following MEX file code creates a persistent array and properly disposes of it.

```
#include "mex.h"

static int initialized = 0;
static mxArray *persistent_array_ptr = NULL;

void cleanup(void) {
    mexPrintf("MEX file is terminating, destroying array\n");
    mxDestroyArray(persistent_array_ptr);
}

void mexFunction(int nlhs,
                 mxArray *plhs[],
                 int nrhs,
                 const mxArray *prhs[])
{
    if (!initialized) {
        mexPrintf("MEX file initializing, creating array\n");

        /* Create persistent array and register its cleanup. */
        persistent_array_ptr = mxCreateDoubleMatrix(1, 1, mxREAL);
        mexMakeArrayPersistent(persistent_array_ptr);
        mexAtExit(cleanup);
        initialized = 1;

        /* Set the data of the array to some interesting value. */
        *mxGetDoubles(persistent_array_ptr) = 1.0;
    } else {
        mexPrintf("MEX file executing; value of first array element is %g\n",
                 *mxGetDoubles(persistent_array_ptr));
    }
}
```

See Also

`mexAtExit` | `mexMakeArrayPersistent`

More About

- “How MATLAB Allocates Memory”
- “Memory Management Issues” on page 7-86

Automatic Cleanup of Temporary Arrays in MEX Files

When a MEX function returns control to MATLAB, it returns the results of its computations in the output arguments—the `mxArrays` contained in the left-side arguments `pLhs[]`. These arrays must have a temporary scope, so do not pass arrays created with the `mexMakeArrayPersistent` function in `pLhs`. MATLAB destroys any `mxArray` created by the MEX function that is not in `pLhs`. MATLAB also frees any memory that was allocated in the MEX function using the `mxMalloc`, `mxRealloc`, or `mxMalloc` functions.

MathWorks recommends that MEX 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. However, there are several circumstances in which the MEX function does not reach its normal return statement.

The normal return is not reached if:

- MATLAB calls `mexCallMATLAB` and the function being called creates an error. (A source MEX file can trap such errors by using the `mexCallMATLABWithTrap` function, but not all MEX files necessarily need to trap errors.)
- The user interrupts the MEX function execution using **Ctrl+C**.
- The MEX function runs out of memory. The MATLAB out-of-memory handler terminates the MEX function.

In the first case, a MEX programmer can ensure safe cleanup of temporary arrays and memory before returning, but not in the last two cases. The automatic cleanup mechanism is necessary to prevent memory leaks in those cases.

You must use the MATLAB-provided functions, such as `mxMalloc` and `mxFree`, to manage memory. Do not use the standard C library counterparts; doing so can produce unexpected results, including program termination.

Example

This example shows how to allocate memory for variables in a MEX function. For example, if the first input to your function (`pRhs[0]`) is a string, to manipulate the string, create a buffer `buf` of size `buflen`. The following statements declare these variables:

```
char *buf;  
int buflen;
```

The size of the buffer depends the number of dimensions of your input array and the size of the data in the array. This statement calculates the size of `bufLen`:

```
bufLen = mxGetN(prhs[0])*sizeof(mxChar)+1;
```

Next, allocate memory for `buf`:

```
buf = mxMalloc(bufLen);
```

At the end of the program, if you do not return `buf` as a `plhs` output parameter, then free its memory as follows:

```
mxFree(buf);
```

Before exiting the MEX function, destroy temporary arrays and free dynamically allocated memory, except if such an `mxArray` is returned in the output argument list, returned by `mxGetVariablePtr`, or used to create a structure. Also, never delete input arguments.

Use `mxFree` to free memory allocated by the `mxMalloc`, `mxMalloc`, or `mxRealloc` functions. Use `mxDestroyArray` to free memory allocated by the `mxCreate*` functions.

See Also

`mxMalloc` | `mxFree`

More About

- “How MATLAB Allocates Memory”
- “Memory Management Issues” on page 7-86

Handling Large File I/O in MEX Files

Prerequisites to Using 64-Bit I/O

MATLAB supports the use of 64-bit file I/O operations in your MEX file programs. You can read and write data to files that are up to and greater than 2 GB (2^{31-1} bytes) in size. Some operating systems or compilers do not support files larger than 2 GB. The following topics describe how to use 64-bit file I/O in your MEX files.

Header File

Header file `io64.h` defines many of the types and functions required for 64-bit file I/O. The statement to include this file must be the *first* `#include` statement in your source file and must also precede any system header include statements:

```
#include "io64.h"
#include "mex.h"
```

Type Declarations

To declare variables used in 64-bit file I/O, use the following types.

MEX Type	Description	POSIX
<code>fpos_T</code>	Declares a 64-bit <code>int</code> type for <code>setFilePos()</code> and <code>getFilePos()</code> . Defined in <code>io64.h</code> .	<code>fpos_t</code>
<code>int64_T</code> , <code>uint64_T</code>	Declares 64-bit signed and unsigned integer types. Defined in <code>tmwtypes.h</code> .	<code>long</code> , <code>long</code>
<code>structStat</code>	Declares a structure to hold the size of a file. Defined in <code>io64.h</code> .	<code>struct stat</code>
<code>FMT64</code>	Used in <code>mexPrintf</code> to specify length within a format specifier such as <code>%d</code> . See example in the section "Printing Formatted Messages" on page 8-74. <code>FMT64</code> is defined in <code>tmwtypes.h</code> .	<code>%lld</code>

MEX Type	Description	POSIX
LL, LLU	Suffixes for literal <code>int</code> constant 64-bit values (C Standard ISO [®] /IEC 9899:1999(E) Section 6.4.4.1). Used only on UNIX systems.	LL, LLU

Functions

Use the following functions for 64-bit file I/O. All are defined in the header file `io64.h`.

Function	Description	POSIX
<code>fileno()</code>	Gets a file descriptor from a file pointer	<code>fileno()</code>
<code>fopen()</code>	Opens the file and obtains the file pointer	<code>fopen()</code>
<code>getFileFstat()</code>	Gets the file size of a given file pointer	<code>fstat()</code>
<code>getFilePos()</code>	Gets the file position for the next I/O	<code>fgetpos()</code>
<code>getFileStat()</code>	Gets the file size of a given file name	<code>stat()</code>
<code>setFilePos()</code>	Sets the file position for the next I/O	<code>fsetpos()</code>

Specifying Constant Literal Values

To assign signed and unsigned 64-bit integer literal values, use type definitions `int64_T` and `uint64_T`.

On UNIX systems, to assign a literal value to an integer variable where the value to be assigned is greater than $2^{31} - 1$ signed, you must suffix the value with `LL`. If the value is greater than $2^{32} - 1$ unsigned, then use `LLU` as the suffix. These suffixes are not valid on Microsoft Windows systems.

Note The `LL` and `LLU` suffixes are not required for hardcoded (literal) values less than 2^G ($2^{31} - 1$), even if they are assigned to a 64-bit `int` type.

The following example declares a 64-bit integer variable initialized with a large literal `int` value, and two 64-bit integer variables:

```

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
#ifdef _MSC_VER || defined(__BORLANDC__)    /* Windows */
    int64_T large_offset_example = 9000222000;
#else                                       /* UNIX */
    int64_T large_offset_example = 9000222000LL;
#endif

    int64_T offset = 0;
    int64_T position = 0;

```

Opening a File

To open a file for reading or writing, use the C/C++ `fopen` function as you normally would. As long as you have included `io64.h` at the start of your program, `fopen` works correctly for large files. No changes at all are required for `fread`, `fwrite`, `fprintf`, `fscanf`, and `fclose`.

The following statements open an existing file for reading and updating in binary mode.

```

fp = fopen(filename, "r+b");
if (NULL == fp)
{
    /* File does not exist. Create new file for writing
     * in binary mode.
     */
    fp = fopen(filename, "wb");
    if (NULL == fp)
    {
        sprintf(str, "Failed to open/create test file '%s'",
                filename);
        mexErrMsgIdAndTxt( "MyToolbox:myfnc:fileCreateError",
                           str);
        return;
    }
    else
    {
        mexPrintf("New test file '%s' created\n",filename);
    }
}
else mexPrintf("Existing test file '%s' opened\n",filename);

```

Printing Formatted Messages

You cannot print 64-bit integers using the %d conversion specifier. Instead, use FMT64 to specify the appropriate format for your platform. FMT64 is defined in the header file `tmwtypes.h`. The following example shows how to print a message showing the size of a large file:

```
int64_T large_offset_example = 9000222000LL;

mexPrintf("Example large file size: %" FMT64 "d bytes.\n",
         large_offset_example);
```

Replacing fseek and ftell with 64-Bit Functions

The ANSI C `fseek` and `ftell` functions are not 64-bit file I/O capable on most platforms. The functions `setFilePos` and `getFilePos`, however, are defined as the corresponding POSIX[®] `fsetpos` and `fgetpos` (or `fsetpos64` and `fgetpos64`) as required by your platform/OS. These functions are 64-bit file I/O capable on all platforms.

The following example shows how to use `setFilePos` instead of `fseek`, and `getFilePos` instead of `ftell`. The example uses `getFileFstat` to find the size of the file. It then uses `setFilePos` to seek to the end of the file to prepare for adding data at the end of the file.

Note Although the offset parameter to `setFilePos` and `getFilePos` is really a pointer to a signed 64-bit integer, `int64_T`, it must be cast to an `fpos_T*`. The `fpos_T` type is defined in `io64.h` as the appropriate `fpos64_t` or `fpos_t`, as required by your platform OS.

```
getFileFstat(fileno(fp), &statbuf);
fileSize = statbuf.st_size;
offset = fileSize;

setFilePos(fp, (fpos_T*) &offset);
getFilePos(fp, (fpos_T*) &position );
```

Unlike `fseek`, `setFilePos` supports only absolute seeking relative to the beginning of the file. If you want to do a relative seek, first call `getFileFstat` to obtain the file size. Then convert the relative offset to an absolute offset that you can pass to `setFilePos`.

Determining the Size of an Open File

To get the size of an open file:

- Refresh the record of the file size stored in memory using `getFilePos` and `setFilePos`.
- Retrieve the size of the file using `getFileFstat`.

Refreshing the File Size Record

Before attempting to retrieve the size of an open file, first refresh the record of the file size residing in memory. If you skip this step on a file that is opened for writing, the file size returned might be incorrect or 0.

To refresh the file size record, seek to any offset in the file using `setFilePos`. If you do not want to change the position of the file pointer, you can seek to the current position in the file. This example obtains the current offset from the start of the file. It then seeks to the current position to update the file size without moving the file pointer.

```
getFilePos( fp, (fpos_T*) &position);
setFilePos( fp, (fpos_T*) &position);
```

Getting the File Size

The `getFileFstat` function takes a file descriptor input argument. Use `fileno` function to get the file pointer of the open file. `getFileFstat` returns the size of that file in bytes in the `st_size` field of a `structStat` structure.

```
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileFstat(fileno(fp), &statbuf))
{
    fileSize = statbuf.st_size;
    mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
}
```

Determining the Size of a Closed File

The `getFileStat` function takes the file name of a closed file as an input argument. `getFileStat` returns the size of the file in bytes in the `st_size` field of a `structStat` structure.

```
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileStat(filename, &statbuf))
{
    fileSize = statbuf.st_size;
    mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
}
```

MinGW-w64 Compiler

You can use the MinGW-w64 compiler to build MEX files and standalone MATLAB engine and MAT-file applications. To install the compiler, use the Add-Ons menu described in “Get and Manage Add-Ons”. Search for MinGW or select from **Features**. For more information, see the File Exchange article “MATLAB Support for MinGW-w64 C/C++ Compiler”.

Building yprime.c Example

You can test the MinGW compiler by building the `yprime.c` example. Copy the source file to a writable folder.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'mex', 'yprime.c'), '.', 'f')
```

If you only have the MinGW compiler installed on your system, the `mex` command automatically chooses MinGW. Go to the next step. However, if you have multiple C or C++ compilers, use `mex -setup` to choose MinGW.

```
mex -setup
```

Build the MEX file.

```
mex yprime.c
```

MATLAB displays a “Building with” message showing what compiler was used to build the MEX file.

Run the function.

```
yprime(1,1:4)
```

For more information, see “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64” on page 8-79.

MinGW Installation Folder Cannot Contain Space

Do not install MinGW in a location with spaces in the path name. For example, do not use:

```
C:\Program Files\mingw-64
```

Instead, use:

C:\mingw-64

Updating MEX Files to Use MinGW Compiler

If you have MEX source files built with a different MATLAB-supported compiler, you might need to modify the code to build with the MinGW compiler. For example:

- Library (.lib) files generated by Microsoft Visual Studio are not compatible with MinGW.
- Object cleanup is not possible when an exception is thrown using the `mexErrMsgIdAndTxt` function from C++ MEX files, resulting in memory leak.
- An uncaught exception in C++ MEX files compiled with MinGW causes MATLAB to crash.
- MEX files with variables containing large amounts of data cannot be compiled, as the compiler runs out of memory.

See Also

More About

- “Add-Ons”
- “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64” on page 8-79

External Websites

- Supported and Compatible Compilers
- <https://www.mathworks.com/matlabcentral/fileexchange/52848-matlab-support-for-mingw-w64-c-c++-compiler>

Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64

Do Not Link to Library Files Compiled with Non-MinGW Compilers

If you use the MinGW compiler to build a MEX file that links to a library compiled with a non-MinGW compiler, such as Microsoft Visual Studio, the file will not run in MATLAB. Library (.lib) files generated by different compilers are not compatible with each other.

You can generate a new library file using the `dlltool` utility from MinGW.

MinGW Installation Folder Cannot Contain Space

Do not install MinGW in a location with spaces in the path name. For example, do not use:

```
C:\Program Files\mingw-64
```

Instead, use:

```
C:\mingw-64
```

MEX Command Does not Choose MinGW

If you only have the MinGW compiler installed on your system, the `mex` command automatically chooses MinGW for both C and C++ MEX files. If you have multiple C or C++ compilers, use `mex -setup` to choose MinGW for both C and, if required, C++ MEX files.

```
mex -setup  
mex -setup cpp
```

If you only type `mex -setup` choosing MinGW, when you compile a C++ file, `mex` might choose a different compiler.

Manually Configure MinGW for MATLAB

When you install MinGW from the MATLAB **Add-Ons** menu, MATLAB automatically detects the MinGW compiler.

If necessary, you can manually configure MinGW, if you have Windows administrative privileges, using the `configuremingw` script. To download this script, see the MATLAB Answers article "I already have MinGW on my computer. How do I configure it to work with MATLAB".

MinGW Behaves Similarly to gcc/g++ on Linux

When modifying compiler flags using the `mex` command, use the Linux compiler flags `CFLAGS` or `CXXFLAGS` instead of the Windows flag `COMPFLAGS`.

Potential Memory Leak Inside C++ MEX Files on Using MEX Exceptions

Error handling in C++ MEX files compiled with the MinGW-w64 compiler is not consistent with MATLAB error handling. If a C++ MEX file contains a class, using the `mexErrMsgIdAndTxt` function to throw a MEX exception can cause a memory leak for objects created for the class.

For example, the following C++ MEX function contains class `MyClass`.

```
#include "mex.h"

class MyClass {
public:
    MyClass() {
        mexPrintf("Constructor called");
    }
    ~MyClass() {
        mexPrintf("Destructor called");
    }
};

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    MyClass X;

    if (nrhs != 0) {
        mexErrMsgIdAndTxt("MATLAB:cppfeature:invalidNumInputs",
            "No input arguments allowed.");
    }
}
```

The MEX function creates object X from MyClass, then checks the number of input arguments. If the MEX function calls mexErrMsgIdAndTxt, the MATLAB error handling does not free memory for object X, thus creating a memory leak.

Unhandled Explicit Exceptions in C++ MEX Files Unexpectedly Terminate MATLAB

If a function in a C++ MEX file throws an explicit exception which is not caught inside the MEX file with a catch statement, then the exception causes MATLAB to terminate instead of propagating the error to the MATLAB command line.

```
#include "mex.h"

class error {}; // Throw an exception of this class

class MyClass
{
    public:
    MyClass(){
        mexPrintf("Constructor called.");
    }
    ~MyClass(){
        mexPrintf("Destructor called.");
    }
};

void doErrorChecking(const MyClass& obj)
{
    // Do error checking
    throw error();
}

void createMyClass()
{
    MyClass myobj;
    doErrorChecking(myobj);
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    createMyClass();
}
```

The MEX function calls `createMyClass`, which creates an object of class `MyClass` and calls function `doErrorChecking`. Function `doErrorChecking` throws an exception of type `error`. This exception, however, is not caught inside the MEX file and causes MATLAB to crash.

This behavior also occurs for classes inheriting from the class `std::exception`.

Workaround

Catch the exception in the MEX function:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    try{
        createMyClass();
    }
    catch(error e){
        // Error handling
    }
}
```

See Also

`mexErrMsgIdAndTxt`

More About

- “MinGW-w64 Compiler” on page 8-77

C++ MEX Applications

- “C++ MEX Functions” on page 9-2
- “Create a C++ MEX Source File” on page 9-5
- “Build C++ MEX Programs” on page 9-10
- “Test Your Build Environment” on page 9-12
- “C++ MEX API” on page 9-14
- “Structure of C++ MEX Function” on page 9-30
- “Managing External Resources from MEX Functions” on page 9-34
- “Handling Inputs and Outputs” on page 9-37
- “Data Access in Typed, Cell, and Structure Arrays” on page 9-41
- “Data Types for Passing MEX Function Data” on page 9-48
- “Call MATLAB Functions from MEX Functions” on page 9-52
- “Catch Exceptions in MEX Function” on page 9-54
- “Execute MATLAB Statements from MEX Function” on page 9-55
- “Set and Get MATLAB Variables from MEX” on page 9-57
- “MATLAB Objects in MEX Functions” on page 9-60
- “Avoid Copies of Arrays in MEX Functions” on page 9-66
- “Displaying Output in MATLAB Command Window” on page 9-69
- “Call MATLAB from Separate Threads in MEX Function” on page 9-71
- “Out-of-Process Execution of C++ MEX Functions” on page 9-75
- “Making async Requests Using mexCallMATLAB” on page 9-84

C++ MEX Functions

In this section...
“C++ MEX API” on page 9-2
“Basic Design of C++ MEX Functions” on page 9-2
“Call MEX Function from MATLAB” on page 9-3
“Examples of C++ MEX Functions” on page 9-3

MEX or MATLAB executable refers to programs that are automatically loaded and can be called like any MATLAB function.

C++ MEX API

C++ MEX functions are based on two C++ APIs:

- The MATLAB Data API supports MATLAB data types and optimizations like copy-on-write for data arrays passed to MEX functions. For more information, see “MATLAB Data API”.
- A subset of the MATLAB C++ Engine API supports calling MATLAB functions, execution of statements in the MATLAB workspace, and access to variables and objects. For more information, see “C++ MEX API” on page 9-14.

The C++ MEX API supports C++11 features and is not compatible with the C MEX API. You cannot mix these APIs in a MEX file.

Basic Design of C++ MEX Functions

A C++ MEX function is implemented as a class named `MexFunction` that inherits from `matlab::mex::Function`. The `MexFunction` class overrides the function call operator, `operator()`. This implementation creates a function object that you can call like a function.

Calling the MEX function from MATLAB instantiates the function object, which maintains its state across subsequent calls to the same MEX function.

Here is the basic design of a C++ MEX function. It is a subclass of `matlab::mex::Function` that must be named `MexFunction`. The `MexFunction` class overrides the function call operator, `operator()`.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        // Function implementation
        ...
    }
};
```

Inputs and outputs to the MEX function are passed as elements in a `matlab::mex::ArgumentList`. Each input or output argument is a `matlab::data::Array` contained in the `matlab::mex::ArgumentList`.

For an example, see “Create a C++ MEX Source File” on page 9-5.

Call MEX Function from MATLAB

To call a MEX function, use the name of the file, without the file extension. The calling syntax depends on the input and output arguments defined by the MEX function. The MEX file must be on the MATLAB path or in the current working folder when called.

Examples of C++ MEX Functions

These examples illustrate the implementation of C++ MEX Functions:

- `arrayProduct.cpp` — Multiplies an array by a scalar input and returns the resulting array.
- `yprime.cpp` — Defines differential equations for restricted three-body problem.
- `phonebook.cpp` — Shows how to manipulate structures.
- `modifyObjectProperty.cpp` — Shows how to work with MATLAB.

See Also

Related Examples

- “Build C++ MEX Programs” on page 9-10
- “C++ MEX API” on page 9-14
- “Structure of C++ MEX Function” on page 9-30

- “C++ MEX Applications”

Create a C++ MEX Source File

Here is how to create a basic C++ MEX function. This function just adds an offset to each element of an input array to demonstrate basic input and output. For a more detailed discussion on creating MEX function source code, see “Structure of C++ MEX Function” on page 9-30 and related topics.

Create Source File

Using your editor, create a file with a .cpp extension and add commented instructions. For example, MyMEXFunction.cpp.

```
/* MyMEXFunction
 * Adds second input to each
 * element of first input
 * a = MyMEXFunction(a,b);
 */
```

Add Required Header Files

For C++ MEX functions, add these header files.

```
/* MyMEXFunction
 * Adds second input to each
 * element of first input
 * a = MyMEXFunction(a,b);
 */
```

```
#include "mex.hpp"
#include "mexAdapter.hpp"
```

Using Convenience Definitions

Optionally specify the namespace for matlab::data and define other conveniences.

```
/* MyMEXFunction
 * Adds second input to each
 * element of first input
 * a = MyMEXFunction(a,b);
 */
```

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
using matlab::mex::ArgumentList;
```

Define MexFunction Class

All C++ MEX functions are implemented as a class named `MexFunction`. This class must derive from `matlab::mex::Function`.

```
/* MyMEXFunction
 * Adds second input to each
 * element of first input
 * a = MyMEXFunction(a,b);
 */

#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {

};
```

Define operator()

All `MexFunction` classes must override the function call operator, `operator()`, to accept two arguments of type `matlab::mex::ArgumentList`. These arguments contain the inputs and outputs.

```
/* MyMEXFunction
 * Adds second input to each
 * element of first input
 * a = MyMEXFunction(a,b);
 */

#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
```

```

using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {

    }

};

```

Add Member Function to Check Arguments

Test to see if the arguments are of the correct type and size. If tests fail, call the MATLAB error function.

```

/* MyMEXFunction
 * Adds second input to each
 * element of first input
 * a = MyMEXFunction(a,b);
 */

#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {

    }

    void checkArguments(ArgumentList outputs, ArgumentList inputs) {
        // Get pointer to engine
        std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();

        // Get array factory
        ArrayFactory factory;

        // Check first input argument
        if (inputs[0].getType() != ArrayType::DOUBLE ||
            inputs[0].getType() == ArrayType::COMPLEX_DOUBLE ||
            inputs[0].getNumberOfElements() != 1)
        {
            matlabPtr->feval(u"error",
                0,
                std::vector<Array>({ factory.createScalar("First input must be scalar double") }));
        }

        // Check second input argument
        if (inputs[1].getType() != ArrayType::DOUBLE ||
            inputs[1].getType() == ArrayType::COMPLEX_DOUBLE)
        {
            matlabPtr->feval(u"error",
                0,
                std::vector<Array>({ factory.createScalar("Input must be double array") }));
        }
    }
};

```

```
    }  
    // Check number of outputs  
    if (outputs.size() > 1) {  
        matlabPtr->feval(u"error",  
            0,  
            std::vector<Array>({ factory.createScalar("Only one output is returned") }));  
    }  
} }  
};
```

Implement Computation

Get the scalar offset and assign it to a const double. Get the input array and move it to a `matlab::data::TypedArray<double>` to work on the array. Add the offset to each element in the array and assign the modified array to the output variable.

```
/* MyMEXFunction  
 * Adds second input to each  
 * element of first input  
 * a = MyMEXFunction(a,b);  
 */  
  
#include "mex.hpp"  
#include "mexAdapter.hpp"  
  
using namespace matlab::data;  
using matlab::mex::ArgumentList;  
  
class MexFunction : public matlab::mex::Function {  
public:  
    void operator()(ArgumentList outputs, ArgumentList inputs) {  
        checkArguments(outputs, inputs);  
        const double offSet = inputs[0][0];  
        TypedArray<double> doubleArray = std::move(inputs[1]);  
        for (auto& elem : doubleArray) {  
            elem += offSet;  
        }  
        outputs[0] = doubleArray;  
    }  
  
    void checkArguments(ArgumentList outputs, ArgumentList inputs) {  
        // Get pointer to engine  
        std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();  
  
        // Get array factory  
        ArrayFactory factory;  
  
        // Check first input argument  
        if (inputs[0].getType() != ArrayType::DOUBLE ||  
            inputs[0].getType() == ArrayType::COMPLEX_DOUBLE ||  
            inputs[0].getNumberOfElements() != 1)  
        {  
            matlabPtr->feval(u"error",  
                0,  
                std::vector<Array>({ factory.createScalar("First input must be scalar double") }));  
        }  
  
        // Check second input argument  
        if (inputs[1].getType() != ArrayType::DOUBLE ||
```



```
        inputs[1].getType() == ArrayType::COMPLEX_DOUBLE)
    {
        matlabPtr->feval(u"error",
            0,
            std::vector<Array>({ factory.createScalar("Input must be double array") }));
    }
    // Check number of outputs
    if (outputs.size() > 1) {
        matlabPtr->feval(u"error",
            0,
            std::vector<Array>({ factory.createScalar("Only one output is returned") }));
    }
};
```

Setup and Build

After installing a supported compiler, use the `mex` command to build your MEX function.

```
mex -setup c++
mex MyMEXFunction.cpp
```

For more information on building MEX functions, see “Build C++ MEX Programs” on page 9-10.

Call MEX Function

Call your MEX function from MATLAB.

```
b = MyMEXFunction(11.5, rand(1000));
```

See Also

`matlab::mex::ArgumentList` | `matlab::mex::Function` | `mex`

Related Examples

- “Build C++ MEX Programs” on page 9-10
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Build C++ MEX Programs

In this section...

“Supported Compilers” on page 9-10

“Build .cpp File with mex Command” on page 9-10

“MEX Include Files” on page 9-10

“File Extensions” on page 9-11

Build your C++ MEX application using the MATLAB `mex` to set up your environment and compile the C++ source code.

Supported Compilers

Use compilers that support C++11. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.

Build .cpp File with mex Command

If you have installed one of the supported compilers, set up the compiler for C++ MEX applications using the `mex` command. When provided with an option to select a compiler, select an installed compiler that the MATLAB C++ MEX supports.

```
mex -setup C++
```

Build your C++ MEX program using the MATLAB `mex` command.

```
mex MyMEXCode.cpp
```

MEX Include Files

Include the following header files in your C++ source code. Header files contain function declarations with prototypes for the routines that you access in the API libraries. These files are in the `matlabroot/extern/include` folder and are the same for Windows, Mac, and Linux systems. C++ MEX files use these header files:

- `mex.hpp` — Definitions for the C++ MEX API
- `mexAdapter.hpp` — Utilities required by the C++ MEX function operator

Note Include `mexAdapter.hpp` only once with the `MexFunction` class definition in MEX applications that span multiple files.

File Extensions

MEX files are platform-specific. MATLAB identifies MEX files by platform-specific extensions. The following table lists the platform-specific extensions for MEX files.

MEX File Platform-Dependent Extension

Platform	Binary MEX File Extension
Linux (64-bit)	<code>mexa64</code>
Apple Mac (64-bit)	<code>mexmaci64</code>
Windows (64-bit)	<code>mexw64</code>

See Also

`matlab::engine::MATLABEngine | mex`

Related Examples

- “C++ Engine API” on page 14-5
- “Test Your Build Environment” on page 14-12
- “C++ MEX Applications”

Test Your Build Environment

To test your installation and environment, build the `arrayProduct.cpp` MEX file that is included with MATLAB. The source code for this MEX example is in subfolders of `matlabroot/extern/examples/cpp_mex`. Be sure that you have a MATLAB-supported compiler installed.

To build a code example, first copy the file to a writable folder on your path. You can use the `arrayProduct.cpp` example to test your build environment. This MATLAB command copies the `arrayProduct.cpp` file to your current folder.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'cpp_mex', 'arrayProduct.cpp'), '.')
```

To set up and build MEX files, use the `mex` command. First set up the environment for C++.

```
mex -setup C++
```

Select the installed compiler that you want to use when prompted by the `mex` setup script. Then call the `mex` command to build your program. Ensure that C++ MEX supports the compiler you select. For an up-to-date list of supported compilers, see [Supported and Compatible Compilers](#).

```
mex arrayProduct.cpp
```

The `mex` command saves the callable file in the current folder.

Call the `arrayProduct` MEX function using the file name without the extension. Pass a scalar multiplier and an array of doubles as arguments. The MEX function multiplies each element in the input array by the multiplier and returns the result.

```
a = arrayProduct(2, [1 2; 3 4])
```

```
a =
```

```
     2     4  
     6     8
```

See Also

[copyfile](#) | [fullfile](#) | [mex](#)

Related Examples

- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

C++ MEX API

Note The C++ MEX API is not compatible with the C MEX API described in “C MEX File Applications”. You cannot mix these APIs in a MEX file.

The C++ MEX API enables you to create applications that take advantage of C++11 features, such as move semantics, exception handling, and memory management.

<code>matlab::mex::Function</code>	Base class for C++ MEX functions
<code>matlab::mex::ArgumentList</code>	Container for inputs and outputs from C++ MEX functions
<code>matlab::engine::MATLABEngine</code>	Class defining engine API

`matlab::mex::Function`

All MEX file implementations are classes that derive from `matlab::mex::Function`.

<code>std::shared_ptr<matlab::engine::MATLABEngine> getEngine()</code>	Get pointer to the MATLABEngine object
<code>void mexLock()</code>	Prevent clearing of MEX file from memory
<code>void mexUnLock()</code>	Allow clearing of MEX file from memory
<code>std::u16string getFunctionName()</code>	Get the name of the current MEX function

`matlab::mex::ArgumentList`

The MEX function arguments passed via the `operator()` of the `MexFunction` class are `matlab::mex::ArgumentList` containers. `ArgumentList` is a full range to the underlying collection of arrays. The `ArgumentList` object supports the following functions.

<code>operator[]</code>	Enables <code>[]</code> indexing into the elements of an <code>ArgumentList</code> .
<code>begin()</code>	Begin iterator.
<code>end()</code>	End iterator.

<code>size()</code>	Returns the number of elements in the argument list. Use this function to check the number of inputs and outputs specified at the call site.
<code>empty()</code>	Returns a logical value indicating if the argument list is empty (<code>size() == 0</code>).

C++ Engine API

Access MATLAB functions, variables, and objects using the `matlab::engine::MATLABEngine` API described in the following sections. To call the `matlab::engine::MATLABEngine` functions, get a shared pointer that is returned by the `matlab::mex::Function::getEngine` function. For example:

```
std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
```

Use this pointer to call engine functions. For example:

```
matlabPtr->feval(...);
```

Call engine functions only on the same thread as the `MexFunction` class.

“ <code>matlab::engine::MATLABEngine::feval</code> ” on page 9-16	Call MATLAB functions with arguments
“ <code>matlab::engine::MATLABEngine::fevalAsync</code> ” on page 9-18	Call MATLAB function with arguments and returned values asynchronously.
“ <code>matlab::engine::MATLABEngine::eval</code> ” on page 9-20	Evaluate MATLAB statements in the base workspace
“ <code>matlab::engine::MATLABEngine::evalAsync</code> ” on page 9-21	Evaluate MATLAB statements in the base workspace asynchronously
“ <code>matlab::engine::MATLABEngine::getVariable</code> ” on page 9-21	Get variables from the MATLAB base or global workspace
“ <code>matlab::engine::MATLABEngine::getVariableAsync</code> ” on page 9-22	Get variables from the MATLAB base or global workspace asynchronously
“ <code>matlab::engine::MATLABEngine::setVariable</code> ” on page 9-23	Put variables in the MATLAB base or global workspace

“matlab::engine::MATLABEngine::setVariableAsync” on page 9-24	Put variables in the MATLAB base or global workspace asynchronously
“matlab::engine::MATLABEngine::getProperty” on page 9-25	Get object property
“matlab::engine::MATLABEngine::getPropertyAsync” on page 9-26	Get object property asynchronously
“matlab::engine::MATLABEngine::setProperty” on page 9-26	Set object property
“matlab::engine::MATLABEngine::setPropertyAsync” on page 9-27	Set object property asynchronously

matlab::engine::MATLABEngine::feval

```
std::vector<matlab::data::Array> feval(const std::u16string &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 std::u16string &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 std::u16string &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 std::u16string &function,
    const std::shared_ptr<matlab::engine::StreamBuffer> &output,
    const std::shared_ptr<matlab::engine::StreamBuffer> &error,
    RhsArgs&&... rhsArgs )
```

```
ResultType feval(const std::u16string &function,
    RhsArgs&&... rhsArgs)
```

Description

Call MATLAB functions from MEX functions. Use `feval` to call MATLAB functions with arguments passed from MEX functions and to return a result from MATLAB to the MEX function.

Inputs and outputs are types defined by the “MATLAB Data API”. There is also a syntax to support native C++ types.

Parameters

<code>const std::u16string &function</code>	Name of the MATLAB function or script to evaluate. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .
<code>const size_t nlhs</code>	Number of returned values.
<code>const std::vector<matlab::data::Array> &args</code>	Multiple input arguments to pass to the MATLAB function in a <code>std::vector</code> . The vector is converted to a column array in MATLAB.
<code>const matlab::data::Array &arg</code>	Single input argument to pass to the MATLAB function.
<code>const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>()</code>	Stream buffer used to store the standard output from the MATLAB function.
<code>const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>()</code>	Stream buffer used to store the error message from the MATLAB function.
<code>RhsArgs&&... rhsArgs</code>	Native C++ data types used for function inputs. <code>feval</code> accepts scalar inputs of these C++ data types: <code>bool</code> , <code>int8_t</code> , <code>int16_t</code> , <code>int32_t</code> , <code>int64_t</code> , <code>uint8_t</code> , <code>uint16_t</code> , <code>uint32_t</code> , <code>uint64_t</code> , <code>float</code> , <code>double</code> .

Return Value

<code>std::vector<matlab::data::Array></code>	Outputs returned from MATLAB function.
<code>matlab::data::Array</code>	Single output returned from MATLAB function.

ResultType	Output returned from MATLAB function as a user-specified type. Can be a <code>std::tuple</code> if returning multiple arguments.
------------	--

Exceptions

`matlab::engine::MATLAB` There is a MATLAB run-time error in the function.
Exception

`matlab::engine::TypeConversionException` The result of a MATLAB function cannot be converted to the specified type.

`matlab::engine::MATLAB` There is a syntax error in the MATLAB function.
SyntaxException

For more information, see “Call MATLAB Functions from MEX Functions” on page 9-52

matlab::engine::MATLABEngine::fevalAsync

```
FutureResult<std::vector<matlab::data::Array>> fevalAsync(const std::u16string &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 std::u16string &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 std::u16string &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 std::u16string &function,
    const std::shared_ptr<matlab::engine::StreamBuffer> &output,
    const std::shared_ptr<matlab::engine::StreamBuffer> &error,
    RhsArgs&&... rhsArgs)

FutureResult<ResultType> fevalAsync(const std::u16string &function,
    RhsArgs&&... rhsArgs)
```

Description

Call MATLAB function with arguments and returned values asynchronously. For more information “Making async Requests Using mexCallMATLAB” on page 9-84.

Parameters

<code>const std::u16string &function</code>	Name of the MATLAB function or script to evaluate. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .
<code>const size_t numReturned</code>	Number of returned values
<code>const std::vector<matlab: :data::Array> &args</code>	Multiple input arguments to pass to the MATLAB function in an <code>std::vector</code> . The vector is converted to a column array in MATLAB.
<code>const matlab::data::Array arg</code>	Single input argument to pass to the MATLAB function.
<code>const std::shared_ptr<mat lab::engine::Stream Buffer> &output = std::shared_ptr<mat lab::engine::Stream Buffer>()</code>	Stream buffer used to store the standard output from the MATLAB function.
<code>const std::shared_ptr<mat lab::engine::Stream Buffer> &error = std::shared_ptr<mat lab::engine::Stream Buffer>()</code>	Stream buffer used to store the error message from the MATLAB function.
<code>RhsArgs&&... rhsArgs</code>	Native C++ data types used for function inputs. <code>feval</code> accepts scalar inputs of these C++ data types: <code>bool</code> , <code>int8_t</code> , <code>int16_t</code> , <code>int32_t</code> , <code>int64_t</code> , <code>uint8_t</code> , <code>uint16_t</code> , <code>uint32_t</code> , <code>uint64_t</code> , <code>float</code> , <code>double</code> .

Return Value

<code>FutureResult</code>	A <code>FutureResult</code> object used to get the result of calling the MATLAB function.
---------------------------	---

Exceptions

None

matlab::engine::MATLABEngine::eval

```
void eval(const std::u16string &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> ())
```

Description

Evaluate a MATLAB statement as a text string in the calling function workspace.

Parameters

```
const std::u16string  MATLAB statement to evaluate  
&statement  
  
const                Stream buffer used to store the standard output from the  
std::shared_ptr<matlab MATLAB statement  
::engine::StreamBuffer  
> &output  
  
const                Stream buffer used to store the error message from the  
std::shared_ptr<matlab MATLAB command  
::engine::StreamBuffer  
> &error
```

Exceptions

```
matlab::engine::MATLA There is a run-time error in the MATLAB statement.  
BExecutionException  
  
matlab::engine::MATLA There is a syntax error in the MATLAB statement.  
BSyntaxException
```

For more information, see “Execute MATLAB Statements from MEX Function” on page 9-55

matlab::engine::MATLABEngine::evalAsync

```
FutureResult<void> evalAsync(const std::u16string &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> ())
```

Description

Evaluate a MATLAB statement as a string asynchronously. For more information “Making async Requests Using mexCallMATLAB” on page 9-84.

Parameters

const std::u16string &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.

Return Value

matlab::engine::FutureResult	A FutureResult object used to wait for the completion of the MATLAB statement.
------------------------------	--

Exceptions

None

matlab::engine::MATLABEngine::getVariable

```
matlab::data::Array getVariable(const std::u16string &varName,
    WorkspaceType workspaceType = WorkspaceType::BASE)
```

Description

Get a variable from the MATLAB base or global workspace.

Parameters

<code>const std::u16string& varName</code>	Name of a variable in the MATLAB workspace. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .
<code>matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE</code>	MATLAB workspace (BASE or GLOBAL) to get the variable from. For more information, see <code>global</code> and <code>matlab::engine::WorkspaceType</code> .

Return Value

<code>matlab::data::Array</code>	Variable obtained from the MATLAB base or global workspace
----------------------------------	--

Exceptions

<code>matlab::engine::MATLABException</code>	The requested variable does not exist in the specified MATLAB base or global workspace.
--	---

For more information, see “Set and Get MATLAB Variables from MEX” on page 9-57

`matlab::engine::MATLABEngine::getVariableAsync`

```
FutureResult<matlab::data::Array> getVariableAsync(const std::u16string &varName,  
WorkspaceType workspaceType = WorkspaceType::BASE)
```

Description

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

Parameters

<code>const std::u16string& varName</code>	Name of the variable in MATLAB workspace. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .
--	--

```
WorkspaceType    MATLAB workspace (BASE or GLOBAL) to get the variable from.
workspaceType = For more information, see global.
WorkspaceType::BASE
SE
```

Return Value

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

Exceptions

None

matlab::engine::MATLABEngine::setVariable

```
void setVariable(const std::u16string &varName,
                const matlab::data::Array &var,
                WorkspaceType workspaceType = WorkspaceType::BASE)
```

Description

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.

Parameters

```
const std::u16string& varName Name of the variable to create in the MATLAB workspace.
Specify the name as an std::u16string. Also, you can specify
this parameter as an std::string.
const matlab::data::Array var Value of the variable to create in the MATLAB workspace.
```

```
matlab::engine::WorkspaceType MATLAB workspace (BASE or GLOBAL) to put the variable into.
orkspaceType For more information, see global and
workspaceType = matlab::engine::WorkspaceType.
matlab::engine::WorkspaceType::BASE
orkspaceType::BASE
E
```

For more information, see “Set and Get MATLAB Variables from MEX” on page 9-57

matlab::engine::MATLABEngine::setVariableAsync

```
FutureResult<void> setVariableAsync(const std::u16string &varName,
    const matlab::data::Array var,
    WorkspaceType workspaceType = WorkspaceType::BASE)
```

Description

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.

Parameters

<code>const std::u16string& varName</code>	Name of the variable to create in the MATLAB workspace. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .
<code>const matlab::data::Array var</code>	Value of the variable to create in the MATLAB workspace
<code>WorkspaceType workspaceType = WorkspaceType::BASE</code>	Put the variable in the MATLAB BASE or GLOBAL workspace. For more information, see <code>global</code> .

Exceptions

None

matlab::engine::MATLABEngine::getProperty

```
matlab::data::Array getProperty(const matlab::data::Array &objectArray,
    size_t index,
    const std::u16string &propertyName)
```

```
matlab::data::Array getProperty(const matlab::data::Array &object,
    const std::u16string &propertyName)
```

Description

Get the value of an object property. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to get.

Parameters

const matlab::data::Array &objectArray	Array of MATLAB objects
const matlab::data::Array &object	Scalar MATLAB object
size_t index	Zero-based index into the object array, specifying the object in that array whose property value is returned
const std::u16string &propertyName	Name of the property. Specify the name as an std::u16string. Also, you can specify this parameter as an std::string.

Return Value

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

Exceptions

```
matlab::engine::MATLABException The property does not exist.
```

For more information, see “MATLAB Objects in MEX Functions” on page 9-60

matlab::engine::MATLABEngine::getPropertyAsync

```
matlab::engine::FutureResult<matlab::data::Array> getPropertyAsync(const matlab::data::Array &objectArray,  
    size_t index,  
    const std::u16string &propertyName)
```

```
matlab::engine::FutureResult<matlab::data::Array> getPropertyAsync(const matlab::data::Array &object,  
    const std::u16string &propertyName)
```

Description

Get the value of an object property asynchronously. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to get.

Parameters

<code>const matlab::data::Array &objectArray</code>	Array of MATLAB objects
<code>const matlab::data::Array &object</code>	Scalar MATLAB object
<code>size_t index</code>	Zero-based index into the object array, specifying the object in that array whose property value is returned
<code>const std::u16string &propertyName</code>	Name of the property. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .

Return Value

```
matlab::engine::FutureResult matlab::engine::FutureResult object that is used to  
    synchronize the operation.
```

Exceptions

None

matlab::engine::MATLABEngine::setProperty

```
void setProperty(matlab::data::Array &objectArray,  
    size_t index,  
    const std::u16string &propertyName,  
    const matlab::data::Array &propertyValue)
```

```
void setProperty(matlab::data::Array &object,
               const std::u16string &propertyName,
               const matlab::data::Array &propertyValue)
```

Description

Set the value of an object property. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to set.

Parameters

<code>matlab::data::Array</code>	Array of MATLAB objects
<code>ay &objectArray</code>	
<code>matlab::data::Array</code>	Scalar MATLAB object
<code>ay &object</code>	
<code>size_t index</code>	Zero-based index into the object array, specifying the object in that array whose property value is set
<code>const std::u16string &propertyName</code>	Name of the property to set. Specify the name as an <code>std::u16string</code> . Also, you can specify this parameter as an <code>std::string</code> .
<code>const matlab::data::Array &propertyValue</code>	Value assigned to the property

Exceptions

<code>matlab::engine::MATLABException</code>	The property does not exist.
--	------------------------------

For more information, see “MATLAB Objects in MEX Functions” on page 9-60

`matlab::engine::MATLABEngine::setPropertyAsync`

```
FutureResult<void> setPropertyAsync(matlab::data::Array &objectArray,
                                   size_t index,
                                   const std::u16string &propertyName,
                                   const matlab::data::Array &propertyValue)
```

```
FutureResult<void> setPropertyAsync(matlab::data::Array &object,
                                   const std::u16string &propertyName,
                                   const matlab::data::Array &propertyValue)
```

Description

Set the value of an object property asynchronously. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to set.

Parameters

```
matlab::data::Array &objects      Array of MATLAB objects
matlab::data::Array &object       Scalar MATLAB object
size_t index                       Zero-based index into the object array, specifying the object in
                                   that array whose property value is set
const std::u16string &propertyName  Name of the property to set. Specify the name as an
                                   std::u16string. Also, you can specify this parameter as an
                                   std::string.
const matlab::data::Array &propertyValue  Value assigned to the property.
```

Exceptions

None

Exception Classes

Exception	Cause
<code>matlab::engine::EngineException</code>	There is a MATLAB run-time error in the function or MATLAB fails to start.
<code>matlab::engine::MATLABSyntaxException</code>	There is a syntax error in the MATLAB function.
<code>matlab::engine::MATLABExecutionException</code>	There is a MATLAB run-time error in the MATLAB function or statement.
<code>matlab::engine::TypeConversionException</code>	The result of the MATLAB function cannot be converted to the specified type

For more information, see “Catch Exceptions in MEX Function” on page 9-54

See Also

Related Examples

- “C++ MEX Applications”

Structure of C++ MEX Function

MEX Function Design

A C++ MEX function is a class that overrides the function call operator, `operator()` to create a function object or functor. These objects behave like MATLAB functions that can accept inputs and return outputs.

Header Files

Include these header files:

- `mex.hpp` — Include this file for the C++ MEX API.
- `mexAdapter.hpp` — Include this file once for the implementation of `MexFunction` class

Namespaces

The C++ MEX APIs are contained in these namespaces:

- `matlab::mex` — MEX interface
- `matlab::data` — MATLAB Data API
- `matlab::engine` — Engine API for C++

Entry Point

Define a C++ MEX function as a class named `MexFunction` that derives from the `matlab::mex::Function` class. The `MexFunction` class overrides the virtual `operator()` of the `matlab::mex::Function` class.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        // check input arguments
        // implement function
        ...
    }
}
```

Passing Arguments

MATLAB passes each input argument as a `matlab::data::Array` in the “`matlab::mex::ArgumentList`” on page 9-14 container. Access each input by indexing into the `ArgumentList` array. For example, `inputs[0]` is the first input, `inputs[1]` is the second, and so on. The number of elements in the `inputs` variable equals the number of input arguments passed to the MEX function when the function is called. `ArgumentList` support iterators and can be used in range-based for loops.

Assign the output arguments to the output variable. For example, `outputs[0]` is the first output assigned, `outputs[1]` is the second, and so on. The number of elements in the `outputs` variable equals the number of outputs assigned when the function is called.

It is often useful to assign input arguments to other types of MATLAB data arrays. Specific types like `matlab::data::TypedArray<T>` or `matlab::data::CharArray` provide additional functionality like iterators and converter functions. Choose the type that matches the type of the input.

For example, the following MEX function assigns the input array to a `matlab::data::TypedArray<double>`. This array type supports the use of a range-based for loop, which is used to multiply each element in the array by 2. The modified array is returned in the `outputs` variable.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {

        // Validate arguments
        checkArguments(outputs, inputs);

        // Implement function
        TypedArray<double> doubleArray = std::move(inputs[0]);
        for (auto& elem : doubleArray) {
            elem *= 2;
        }

        // Assign outputs
        outputs[0] = doubleArray;
    }

    void checkArguments(ArgumentList outputs, ArgumentList inputs) {
        std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
        ArrayFactory factory;
        if (inputs[0].getType() != ArrayType::DOUBLE ||
            inputs[0].getType() == ArrayType::COMPLEX_DOUBLE)
    }
}
```

```
    {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Input must be double array") }));
    }

    if (outputs.size() > 1) {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Only one output is returned") }));
    }
}
};
```

Build and run the MEX function.

```
mex timesTwo.cpp
timesTwo(1:10)
```

```
ans =
```

```
     2     4     6     8    10    12    14    16    18    20
```

For more information on validating arguments, see “Handling Inputs and Outputs” on page 9-37.

Class Constructor and Destructor

Calling the MEX function from MATLAB instantiates the `MexFunction` class. For example, this MATLAB statement creates an instance of the `MexFunction` class defined by the `myMEXFunction.cpp` file.

```
output = myMEXFunction(input);
```

This instance continues to exist until you call the MATLAB `clear mex` command.

Implementing a class constructor and destructor provides a way to perform certain tasks when constructing the `MexFunction` object. For example, this code snippet opens a text file for reading in the constructor and closes the file in the destructor.

```
#include "mex.hpp"
#include "mexAdapter.hpp"
#include <fstream>

using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {
    std::ifstream inFile;
```



```
public:
    MexFunction() {
        inFile.open("someTextFile.txt");
    }

    ~MexFunction() {
        inFile.close();
    }

    void operator()(ArgumentList outputs, ArgumentList inputs) {
        ....
    }
};
```

For an example, see “Managing External Resources from MEX Functions” on page 9-34.

See Also

`matlab::data::Array` | `matlab::data::TypedArray`

Related Examples

- “Avoid Copies of Arrays in MEX Functions” on page 9-66
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Managing External Resources from MEX Functions

MEX functions that connect to external resources like sockets, sensors, files, and databases need to manage these resources. Because you implement C++ MEX functions as classes, you can define constructor and destructor functions to manage these resources and variables that persist across repeated calls to the MEX function.

When you call a MEX function, MATLAB creates an instance of the `MexFunction` class. This object persists for the MATLAB session or until you clear the object with the `clear mex` command. Repeated calls to the MEX function can process incoming data and release resources when finished.

Reading a Text File

This MEX function opens a text file and reads one word each time you call the function. The `MexFunction` class that implements the MEX function defines a constructor and a destructor to open and close the file. Each word read is stored in an `std::unordered_map` to determine the number of times that word occurs in the file.

Constructor

The `MexFunction` constructor performs these steps:

- Call `mexLock` to prevent clearing of the MEX function from memory.
- Get the full path to the text file from MATLAB using `"matlab::engine::MATLABEngine::eval"` on page 9-20 and `"matlab::engine::MATLABEngine::getVariable"` on page 9-21.
- Open the text file to the `std::ifstream` input stream.

Destructor

The class destructor closes the file.

Function Call Operator `operator()`

Each call to the MEX function reads a word from the text file and adds it to the `unordered_map`, or just increments the word count for that word if it exists in the map. The MEX function displays the current word and its count in the MATLAB command window using a `std::ostringstream` output stream. To unlock the MEX function, pass an argument (such as `'unlock'`) to the function.

Display On MATLAB

The `displayOnMATLAB` member function uses “`matlab::engine::MATLABEngine::feval`” on page 9-16 to call the MATLAB `fprintf` function with the string written to the output stream.

Code Listing

```
#include "mex.hpp"
#include "mexAdapter.hpp"
#include <unordered_map>
#include <fstream>

using matlab::mex::ArgumentList;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {

    // Input stream to read words from file
    std::ifstream inFile;

    // Unordered map to keep track of word count
    std::unordered_map<std::string, int> wordCount;

    // Pointer to MATLAB engine
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();

    // Factory to create MATLAB data arrays
    ArrayFactory factory;

public:
    MexFunction() {
        mexLock();
        matlabPtr->eval(u"fname = fullfile(matlabroot, 'examples', 'matlab', 'sonnets.txt');");
        matlab::data::CharArray fileName = matlabPtr->getVariable(u"fname");
        inFile.open(fileName.toAscii());
        if (!inFile.is_open()) {
            std::ostringstream stream;
            stream << "Failed to open sonnets.txt" << std::endl;
            displayOnMATLAB(stream);
        }
    }

    ~MexFunction() {
        if (inFile.is_open())
            inFile.close();
    }

    void operator()(ArgumentList outputs, ArgumentList inputs) {
        if (inFile.is_open() && !inFile.eof()) {
            std::string word;
            inFile >> word;
            wordCount[word]++;
            std::ostringstream stream;
            stream << "Read : " << "\"" << word << "\""
                << ", current count: " << wordCount[word] << std::endl;
            displayOnMATLAB(stream);
        }
        if (!inputs.empty() || !inFile.is_open()) {
            mexUnlock();
        }
    }
};
```

```
    }  
}  
  
void displayOnMATLAB(const std::ostringstream& stream){  
    matlabPtr->feval(u"fprintf", 0,  
        std::vector<Array>({ factory.createScalar(stream.str()) }));  
}  
};
```

Build and Run sonnetWordCount.cpp

Open the source code file, `sonnetWordCount.cpp`, in the editor and use the `mex` command to compile the MEX function.

```
mex sonnetWordCount.cpp
```

Call the MEX function repeatedly to count word usage.

```
>> sonnetWordCount  
Read : "THE", current count: 1  
>> sonnetWordCount  
Read : "SONNETS", current count: 1  
>> sonnetWordCount  
Read : "by", current count: 1  
>> sonnetWordCount  
Read : "William", current count: 1  
>> sonnetWordCount  
Read : "Shakespeare", current count: 1  
>> sonnetWordCount('unlock')
```

See Also

`matlab::mex::ArgumentList`

Related Examples

- “Structure of C++ MEX Function” on page 9-30
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Handling Inputs and Outputs

The C++ MEX API provides features that enable you to access and validate arguments. Both `matlab::mex::ArgumentList` and `matlab::data::Array` define functions that are useful to perform checks on the arguments passed to the MEX function. The `matlab::data::ArrayType` enumeration class enables you to test for specific types of inputs.

Argument Validation

This table shows functions useful for error checking.

<code>matlab::mex::ArgumentList::size()</code>	Determine number of inputs. The size of the <code>ArgumentList</code> array is equal to the number of arguments used in the function call.
<code>matlab::mex::ArgumentList::empty()</code>	Test for no inputs or outputs. If the MEX function is called with no input or output arguments, then the respective <code>ArgumentList</code> argument is empty.
<code>matlab::data::Array::getType()</code>	Determine the type of an input argument. This function returns a <code>matlab::data::ArrayType</code> object.
<code>matlab::data::Array::getDimensions()</code>	Determine the dimensions of an input argument array. This function returns a <code>matlab::data::ArrayDimensions</code> object, which is defined as <code>std::vector<size_t></code> .
<code>matlab::data::Array::getNumberOfElements()</code>	Determine the number of elements in an input argument array.
<code>matlab::data::Array::isEmpty()</code>	Determine if an input argument array is empty.

These examples show how to test argument values in your MEX file and throw errors in MATLAB when tests fail. Use the “`matlab::engine::MATLABEngine::feval`” on page 9-16 function to call the MATLAB error or warning functions.

Call `feval` with these arguments:

- The `error` or `warning` function name, passed as a UTF16 string.
- The number of returned arguments, which is zero in these examples.
- The messages to display with the MATLAB `error` or `warning` functions. Pass the message in an `std::vector` containing the `matlab::data::Array` created using the `matlab::data::ArrayFactory` factory.

To call `feval` and define the arguments, get a pointer to a MATLAB engine and a MATLAB data array factory.

```
class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
        matlab::data::ArrayFactory factory;
        ...
    }
}
```

MEX file requires three input arguments.

```
if (inputs.size() != 3) {
    matlabPtr->feval(u"error", 0,
        std::vector<matlab::data::Array>({ factory.createScalar("Three inputs required") }));
}
```

MEX file requires the assignment of two output arguments.

```
if (outputs.size() != 2) {
    matlabPtr->feval(u"error", 0,
        std::vector<matlab::data::Array>({ factory.createScalar("Two outputs required") }));
}
```

Second input argument must be a 1-by-4 row vector of noncomplex doubles.

```
if (inputs[1].getType() != matlab::data::ArrayType::DOUBLE ||
    inputs[1].getType() == matlab::data::ArrayType::COMPLEX_DOUBLE ||
    inputs[1].getNumberOfElements() != 4 ||
    inputs[1].getDimensions()[1] == 1) {
    matlabPtr->feval(u"error", 0,
        std::vector<matlab::data::Array>({ factory.createScalar("Input must be 4-element row vector") }));
}
```

ArgumentList Iterators

The `matlab::mex::ArgumentList` container implements a pair of iterators that mark the beginning and end of the argument list. Use these iterators to loop over all the inputs to a MEX function. For example, this MEX function sums all its inputs using a range-based `for` loop.

```
#include "mex.hpp"
#include "mexAdapter.hpp"
```

```

class MexFunction : public matlab::mex::Function {
    matlab::data::ArrayFactory factory;
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        double sm = 0;
        for (const matlab::data::TypedArray<double>& elem : inputs) {
            sm = sm + elem[0];
        }
        outputs[0] = factory.createScalar(sm);
    };
};

```

Save this MEX file as `sumInputs.cpp` and compile with the `mex` command. Call the function with a range of arguments.

```

mex sumInputs.cpp
sumInputs(1,2,3,4,5,6,7)

```

```

ans =

```

```

    28

```

Support Variable Number of Arguments

You can support variable number of arguments in MEX functions by checking the size of the input and output `matlab::mex::ArgumentList` array.

In the following code snippet, the size of the `outputs` parameter array indicates how many outputs are specified when the MEX function is called. Using the `ArgumentList::size` function, this code determines how many outputs are specified when the MEX function is called.

```

void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {

    if (outputs.size() == 3) {
        outputs[0] = // Assign first output
        outputs[1] = // Assign second output
        outputs[2] = // Assign third output
    }
    else if (outputs.size() == 1) {
        outputs[0] = // Assign only one output
    }
    else {
        matlabPtr->feval(u"error", 0,
            std::vector<matlab::data::Array>({ factory.createScalar("One or three outputs required")}));
    }
}

```

See Also

`matlab::mex::ArgumentList`

Related Examples

- “C++ MEX API” on page 9-14
- “MATLAB Data API”
- “C++ MEX Applications”

Data Access in Typed, Cell, and Structure Arrays

The MATLAB Data API used by C++ MEX functions provides the same copy-on-write semantics used by functions written directly in MATLAB. A MEX function can duplicate a data array passed to it, but this duplicate is a shared copy of the original variable. A separate copy of the data is made when the MEX function modifies values in the array.

When you call a MEX function, the `matlab::data::Array` inputs are shared copies of the MATLAB variables passed to the MEX function. Shared copies provide an advantage when the MEX function:

- Does not modify a large array passed to the MEX function.
- Modifies certain fields of a structure that is passed as input without making a copy of the entire structure.
- Modifies a cell in a cell array that is passed as input without causing a deep copy of the cell array
- Modifies an object property that is passed as input without making a deep copy of the object.

Shared Copies

This example code benefits from the use of shared copies when calculating the average value of an image array. Copying the input array to a `const matlab::data::TypedArray<T>`, which supports iterators, enables the use of a range-based for loop to perform the calculation, but ensures that the array is not copied.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
    ArrayFactory factory;
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        double sm = 0;
        const TypedArray<uint8_t> inArray = inputs[0];
        for (auto& elem : inArray) {
            sm += elem;
        }
        outputs[0] = factory.createScalar(sm / inArray.getNumber0fElements());
    }
};
```

You can use this function to find the mean value of an array of topographic data. Load the `topo.mat` file into the MATLAB workspace and pass the data array to the MEX function. Save the MEX function source code in a file called `aveImage.cpp`.

```
mex aveImage.cpp
load topo
m = aveImage(topo)

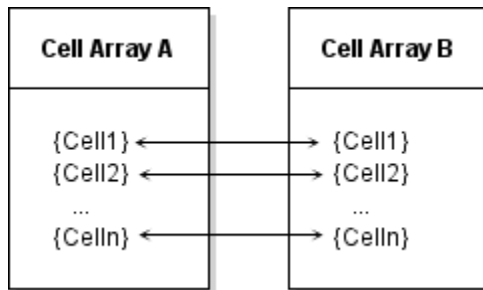
m =

    -1.8948e+03
```

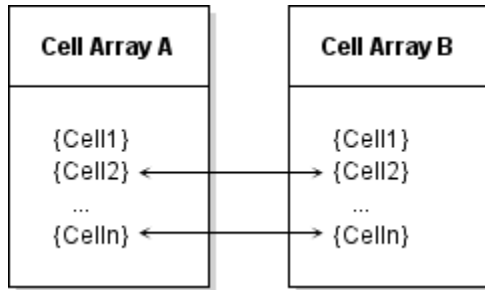
Modify Cell Array in MEX Function

When you pass a cell array to a MEX function, its MATLAB Data API representation, `matlab::data::CellArray`, is essentially a `matlab::data::Array` that contains a `matlab::data::Array` in each cell. This design enables each contained array to be a shared copy until modified.

When you modify a cell of a cell array in a MEX function, only the array contained in that cell is unshared, not the entire cell array. For example, suppose that you copy cell array A to cell array B. The value of each cell is shared.



If you modify `Cell11` in cell array B, the array in that cell is no longer shared with `Cell11` in cell array A. However the values of all other cells remain shared so that the entire cell array does not need to be copied.



This example shows how to pass a cell array to a MEX function and modify one cell in a cell array without causing copies of the data in the other cells. The example loads image data into the MATLAB workspace and creates a cell array that contains the image data, caption, and colormap.

```
load durer
whos
```

Name	Size	Bytes	Class
X	648x509	2638656	double
caption	2x28	112	char
map	128x3	3072	double

Assign each variable to a different cell.

```
durerCell{1} = X;
durerCell{2} = caption;
durerCell{3} = map;
```

This MEX function uses `std::move` to move the input cell array to a `matlab::data::CellArray`, which provides access to individual cells. The only data modified is the caption that is stored in one cell of the cell array. To modify that cell:

- Get a `matlab::data::Reference<TypedArray<T>>` reference to the array in the cell containing the caption.
- Assign the new value to the reference as a `matlab::data::CharArray` because the returned value is a MATLAB character vector.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::data;
```

```
class MexFunction : public matlab::mex::Function {
    ArrayFactory factory;
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        checkArguments(inputs);
        CellArray imageCell = std::move(inputs[0]);
        TypedArrayRef<char16_t> cellRef = imageCell[1];
        cellRef = factory.createCharArray("Albrecht Durer's Melancholia with a magic square");
        outputs[0] = imageCell;
    }

    void checkArguments(ArgumentList inputs) {
        std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
        if (inputs[0].getType() != ArrayType::CELL) {
            matlabPtr->feval(u"error", 0, std::vector<Array>
                ({ factory.createScalar("Input must be cell array") }));
        }
    }
};
```

Save this code in a file named `modifyCellArray.cpp`, build the MEX file, and call the function.

```
mex modifyCellArray.cpp
durerCell = modifyCellArray(durerCell);
```

The returned cell array contains the new character array assigned in the MEX function.

```
durerCell{2}
```

```
ans =
```

```
    'Albrecht Durer's Melancholia with a magic square'
```

For more information on working with cell arrays, see “C++ Cell Arrays” on page 2-7.

Modify Structure in MEX Function

The MATLAB Data API defines `matlab::data::StructArray` to represent MATLAB struct arrays. Each field in a `StructArray` itself contains an array. Therefore, you can modify a field of a `StructArray` passed to a MEX function without causing the whole array to be copied. Fields not modified remain shared with the input array fields.

The `StructArray` class provides member functions to access the fields of a structure:

- `getFieldNames` returns begin and end iterators providing access to field names.
- `getNumberOfFields` returns the number of fields in the structure.

For example, you can generate a vector of field names and use these names to access the data in the structure. This code assumes that the field contains an array of type `double`.

```
auto fields = myStruct.getFieldNames();
std::vector<matlab::data::MATLABFieldIdentifier> fieldNames(fields.begin(), fields.end());
// Get the data from one field
matlab::data::TypedArray<double> field1 = myStruct[0][fieldNames[0]]
```

Assign new values to a field by creating an array of the correct type.

```
myStruct[0][fieldNames[0]] = factory.createArray<double>({ 1,5 }, { 1, 2, 3, 4, 5 });
```

Create a reference to a field of a structure using `matlab::data::Reference<T>`, which you can pass to functions that access the values in the field. References enable you to get the value of a field, to assign a new value to the field, and to create another array that refers to the same value. For example, this code creates a reference to a field containing an array of doubles.

```
auto fields = myStruct.getFieldNames();
std::vector<matlab::data::MATLABFieldIdentifier> fieldNames(fields.begin(), fields.end());
matlab::data::TypedArrayRef<double> field1Reference = myStruct[0][fieldNames[0]]
```

Assign new values to the field using this reference.

```
field1Reference = factory.createArray<double>({ 1,5 }, { 1, 2, 3, 4, 5 });
```

Assign to Structure Field

This example passes a MATLAB structure to a MEX function. The structure contains two large data fields, and two fields to contain scalar values assigned by the MEX function. The MEX function calculates the average value for the numbers in each array and assigns these values to the `Average` field of the respective structure.

This MATLAB code creates the structure array and passes it to the MEX function built from the `modifyStruct.cpp` file described here.

```
s = struct('Average', {[], []}, ...
         'Data', {rand(1,1000), randi([1,9], 1,1000)});
s = modifyStruct(s);
```

Here is the `MexFunction::operator()` function. It performs these operations:

- Call the `checkArgument` function to check the inputs and outputs for size and type.
- Assign the input to a `matlab::data::StructArray` variable.
- Call the `calcMean` function to compute the averages.
- Assign the updated structure array to the MEX function output.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        checkArguments(outputs, inputs);
        StructArray inStruct(inputs[0]);
        calcMean(inStruct);
        outputs[0] = inStruct;
    }
}
```

The `checkArguments` function performs these checks:

- The number of inputs equals one.
- The number of outputs is not greater than one.
- The input is a MATLAB structure array.

```
void checkArguments(ArgumentList outputs, ArgumentList inputs) {
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
    ArrayFactory factory;
    if (inputs.size() != 1) {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("One input required") }));
    }
    if (outputs.size() > 1) {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Too many outputs specified") }));
    }
    if (inputs[0].getType() != ArrayType::STRUCT) {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Input must be structure") }));
    }
}
```

The `calcMean` function calculates the average value for each set of numbers in the `Data` fields and assigns these values to the `Average` field of the respective structure.

```
void calcMean(StructArray& inStruct) {
    ArrayFactory factory;
    auto fields = inStruct.getFieldNames();
    std::vector<MATLABFieldIdentifier> fieldNames(fields.begin(), fields.end());
    double sm = 0;
    for (auto i = 0; i < 2; i++) {
        const TypedArray<double> data = inStruct[i][fieldNames[1]];
        for (auto& elem : data) {
            sm += elem;
        }
        inStruct[i][fieldNames[0]] = factory.createScalar(sm / data.getNumberofElements());
    }
}
```

```
}; } }
```

Example of Cell and Structure Arrays

For a related example that uses cell and structure arrays, open this source file in the MATLAB editor `phonebook.cpp`.

See Also

`matlab::data::MATLABFieldIdentifier`

Related Examples

- “MATLAB Data API”
- “Create Cell Arrays from C++” on page 14-41
- “Create Structure Arrays from C++” on page 14-46
- “C++ MEX Applications”

Data Types for Passing MEX Function Data

The MATLAB Data API supports array types that enable MEX functions to pass specific data types from and to MATLAB. For information on additional array types, see “MATLAB Data API”.

The most general type of array is the `matlab::data::Array`. More specific types provide additional functionality. For example, `matlab::data::TypedArray<T>` provides iterator support and `matlab::data::CharArray` provides converters for ASCII and UTF16 types.

The following sections show how to define input and output types using the MATLAB Data API. Assume a MEX framework with inputs and outputs defined as shown in the following class definition. Use the `matlab::data::ArrayFactory` to create output arrays.

```
class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        matlab::data::ArrayFactory factory;
        ...
    }
};
```

Typed Arrays

Use `matlab::data::TypedArray<T>` to define specific types, such as numeric and logical values. For example, call `myMexFcn` from MATLAB.

```
m = [-2 2 6 8];
result = myMexFcn(m);
```

Assign input of MATLAB type `double` in the MEX function.

```
matlab::data::TypedArray<double> doubleArray = inputs[0];
```

Return output to be of type `double` in MATLAB:

```
outputs[0] = factory.createArray<double>({ 1,4 }, { -2.0, 2.0, 6.0, 8.0 });
```

Character Arrays

Use `matlab::data::CharArray` to pass character arrays to and from MEX functions. For example, call `myMexFcn` from MATLAB with a character vector.

```
result = myMexFcn('Character vector');
```


Assign input of MATLAB type `char` in the MEX function.

```
matlab::data::CharArray charVector2 = inputs[0];
```

Return output to be of type `char` in MATLAB.

```
outputs[0] = factory.createCharArray("Character vector");
```

String Arrays

Use `matlab::data::TypedArray<MATLABString>` to pass string arrays to and from MEX functions. For example, call `myMexFcn` from MATLAB with a string array.

```
result = myMexFcn(["Array", "of", "strings"]);
```

Assign input of MATLAB type `string` in the MEX function.

```
matlab::data::TypedArray<matlab::data::MATLABString> stringArray = inputs[0];
```

Return output to be of type `string` in MATLAB.

```
outputs[0] = factory.createArray({ 1,3 }, { u"Array", u"of", u"strings" });
```

Cell Arrays

Use `matlab::data::CellArray` to pass cell arrays to and from MEX functions. For example, call `myMexFcn` from MATLAB with a cell array.

```
result = myMexFcn({'MATLAB cell array', [1.2 2.2; 3.2 4.2]});
```

Assign input of MATLAB type `cell` in the MEX function.

```
matlab::data::CellArray inCellArray2 = inputs[0];
```

Return output to be of type `cell` in MATLAB.

```
outputs[0] = factory.createCellArray({ 1,2 },
    factory.createCharArray("MATLAB Cell Array"),
    factory.createArray<double>({ 2,2 }, { 1.2, 3.2, 2.2, 4.2 }));
```

Note the row-major vs. column-major ordering difference between C++ and MATLAB when defining 2-D arrays.

Structure Arrays

Use `matlab::data::StructArray` to pass structures to and from MEX functions. For example, call `myMexFcn` from MATLAB with a structure.

```
st.Name = 'Color';
st.Value = uint8([1 0 1]);
result = myMexFcn(st);
```

Assign input of MATLAB type `struct` in the MEX function.

```
matlab::data::StructArray inStructArray = inputs[0];
```

Return output to be of type `struct` in MATLAB.

```
matlab::data::StructArray S = factory.createStructArray({ 1,1 }, { "Name","Value" });
S[0]["Name"] = factory.createCharArray("Color");
S[0]["Value"] = factory.createArray<uint8_t>({ 1, 3 }, { 1, 0, 1 });
outputs[0] = S;
```

MATLAB Objects

Use `matlab::data::Array` to pass objects to and from MEX functions. For example, call `myMexFcn` from MATLAB with an object of the user-defined class named `MyClass`.

```
classdef MyClass
    property
        MeanValue = 0.5
    end
end
```

```
obj = MyClass;
```

Assign input of MATLAB type `MyClass` in the MEX function.

```
matlab::data::Array obj = inputs[0];
```

Assume that `MyClass` defines a property called `MeanValue` that contains a scalar `double`. Get the property value using

`matlab::engine::MATLABEngine::getProperty`.

```
matlab::data::TypedArray<double> meanValue = matlabPtr->getProperty(obj, u"MeanValue");
double m = meanValue[0];
```

Set the property value using `matlab::engine::MATLABEngine::setProperty` and return the object to MATLAB as a `matlab::data::Array`.

```
matlabPtr->setProperty(obj, u"MeanValue", factory.createScalar<double>(1.02));  
outputs[0] = obj;
```

For an example of how to work with MATLAB objects, see “MATLAB Objects in MEX Functions” on page 9-60.

See Also

cell | char | double | string | struct

Related Examples

- “MATLAB Data API”
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Call MATLAB Functions from MEX Functions

Call MATLAB functions from MEX functions using the “`matlab::engine::MATLABEngine::feval`” on page 9-16 function. `feval` enables you to pass arguments from MEX functions to MATLAB functions and to return the results to the MEX function.

The following code snippets require these definitions to use the `matlab::data::ArrayFactory` and the “C++ Engine API” on page 9-15.

```
matlab::data::ArrayFactory factory;  
std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
```

Single Output

This example calls the MATLAB `sqrt` function with the following inputs:

- The function name, passed as a UTF16 string
- The inputs to the `sqrt` function, specified as a `matlab::data::Array`

The value returned to the MEX function is a four-element `matlab::data::Array` containing the square root of each element of the input array.

The example moves the returned value to a `matlab::data::TypedArray`, which provides iterators used in the range-based `for` loop that creates an array of type `double` from the results returned by the MATLAB `sqrt` function.

```
// Define input and output arguments  
matlab::data::Array args({  
    factory.createArray<double>({ 1, 4 },{ 1, 2, 3, 4 }) });  
matlab::data::Array result;  
  
// Call feval and return 1 argument  
result = matlabPtr->feval(u"sqrt", args);  
matlab::data::TypedArray<double> returnedValues(std::move(result));  
  
// Create native array  
double dataArray[4];  
int i = 0;  
for (auto elem : returnedValues) {  
    dataArray[i] = elem;  
    i++;  
}
```

Multiple Outputs

Some MATLAB functions return different numbers of outputs depending on how you call the function. You can specify the number of returned arguments when calling a MATLAB function from a MEX function.

This code calls the MATLAB `gcd` function with the following inputs:

- The function name passed as a UTF16 string
- The number of outputs returned by the MATLAB function, specified as a `const size_t`.
- The inputs to the `gcd` function, specified as a `std::vector` of `matlab::data::Array` elements.

The returned value is a `std::vector` containing three `matlab::data::Array` elements.

```
// Define arguments
std::vector<matlab::data::Array> args({
    factory.createScalar<int16_t>(30),
    factory.createScalar<int16_t>(56)});
const size_t numReturned = 3;
std::vector<matlab::data::Array> result;

// Call feval and return 3 arguments
result = matlabPtr->feval(u"gcd", numReturned, args);
```

See Also

“`matlab::engine::MATLABEngine::feval`” on page 9-16

Related Examples

- “Execute MATLAB Statements from MEX Function” on page 9-55
- “MATLAB Data API”
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Catch Exceptions in MEX Function

To override the default error behavior, you can catch exceptions thrown in MEX functions when calling MATLAB functions.

This code causes MATLAB to throw an exception because it defines the input argument incorrectly for the MATLAB `sqrt` function. The catch block handles the `matlab::engine::MATLABException` by displaying the string describing the exception in the MATLAB command window.

```
ArrayFactory factory;
std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();

// Variable with no value causes error
std::vector<matlab::data::Array> arg;
try {
    matlab::data::Array result =
        matlabPtr->feval(u"sqrt", arg);
}
catch (const matlab::engine::MATLABException& ex) {
    matlabPtr->feval(u"disp", 0, std::vector<Array>({factory.createScalar(ex.what()) }));
}
```

See Also

Related Examples

- “Handling Inputs and Outputs” on page 9-37
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Execute MATLAB Statements from MEX Function

MEX functions can execute MATLAB statements in the calling function workspace. Evaluating statements in the workspace of the calling function enables MEX functions to create or modify variables in the workspace in which it executes. MEX functions can also modify the environment, for instance by changing the current folder.

The MATLAB statements can access any variables that are in scope in the calling function workspace. If the MEX function is called from the MATLAB base workspace, then the statements are evaluated in the context of that workspace.

To execute MATLAB statements from a MEX function, use the “`matlab::engine::MATLABEngine::eval`” on page 9-20 function. Use `eval` when you do not need to pass arguments to a MATLAB function or return arguments to the MEX function.

Pass the MATLAB statement to `eval` as an `std::u16string`. Use the `u"..."` UTF-16 literal string encoding or the utility function `matlab::engine::convertUTF8StringToUTF16String` to convert an `std::string` to an `std::u16string`. The functions and input arguments named in the string must exist in the caller's workspace.

This code snippet shows how to use `eval` to execute MATLAB statements. These statements add the current folder to the MATLAB path and then change the MATLAB working folder to the one mapped to drive H on a Windows system. Note the escape (“\”) character in front of the backslash character.

```
std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
matlabPtr->eval(u"currentFolder = pwd;");
matlabPtr->eval(u"addpath(currentFolder);");
matlabPtr->eval(u"cd('H:\\')");
```

Here is the equivalent MATLAB code.

```
currentFolder = pwd;
addpath(currentFolder);
cd('H:\\')
```

See Also

“`matlab::engine::MATLABEngine::feval`” on page 9-16 | `addpath` | `cd` | `pwd`

Related Examples

- “Call MATLAB Functions from MEX Functions” on page 9-52
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

Set and Get MATLAB Variables from MEX

MEX functions can put variables into the MATLAB base and global workspaces during MEX function execution. MEX functions can get variables from the MATLAB base and global workspaces during MEX function execution.

To put variables in the MATLAB base or global workspace from MEX functions use the “`matlab::engine::MATLABEngine::setVariable`” on page 9-23 function.

To get variables from the MATLAB base or global workspace and bring them into a MEX function, use the “`matlab::engine::MATLABEngine::getVariable`” on page 9-21 function.

Get Variable from MATLAB Workspace

Suppose that there is a variable named `result` in the MATLAB base workspace. This variable is of type `double`.

```
result = 1^3 + 5^3 + 3^3;
```

To get the `result` variable from the MATLAB workspace, call `getVariable`, which returns the variable as a `matlab::data::Array`.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::engine;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
    ArrayFactory factory;
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        Array result = matlabPtr->getVariable(u"result");
        double mexResult = std::move(result[0]);
    }
};
```

The `result` variable is a shared copy of the `result` variable in the MATLAB workspace. Using `std::move` when assigning `result` in the MEX function to the native `double` variable `mexResult` unshares the value with the variable in the MATLAB workspace.

Put Variable in MATLAB Workspace

MEX functions can put variables in the MATLAB base or global workspace. If a variable with the same name exists in the specified workspace, `setVariable` overwrites it.

For example, you can make a variable available from the MATLAB global workspace so that any MATLAB function can define this global variable.

This MEX function creates a global variable called `mexGlobal`. The value of this variable is 153.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::engine;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
    ArrayFactory factory;
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        Array val = factory.createScalar(153.0);
        matlabPtr->setVariable(u"mexGlobal", val, WorkspaceType::GLOBAL);
    }
};
```

To access the global variable from MATLAB functions, use the `global` keyword to define the variable as global in the function workspace.

```
function testMexGlobal
    global mexGlobal
    localVar = 1^3 + 5^3 + 3^3;
    if localVar == mexGlobal
        disp('Global found')
    end
end
```

Global variables are shared among all functions that declare the variable as `global`. Any change of value to that variable, in any function, is visible to all the functions that declare it as `global`.

Set and Get Variable Example Code

For a complete example that uses “`matlab::engine::MATLABEngine::getVariable`” on page 9-21, download these two files and follow the instructions in the files to build and run the MEX function.

`mexgetarray.cpp` and `mexgetarray.hpp`.

See Also

global

Related Examples

- “MATLAB Objects in MEX Functions” on page 9-60
- “Global Variables”
- “C++ MEX API” on page 9-14
- “C++ MEX Applications”

MATLAB Objects in MEX Functions

MEX functions can access MATLAB object properties using the “`matlab::engine::MATLABEngine::setProperty`” on page 9-26 and “`matlab::engine::MATLABEngine::getProperty`” on page 9-25 member functions. Objects passed to MEX functions behave like objects passed to any MATLAB function:

- Handle objects - Changes made to handle objects in MEX functions affect the object in the caller's workspace.
- Value objects - Changes made to value objects in MEX functions affect only the independent copy of the object in the workspace of the MEX function.

Therefore, value objects modified in MEX functions must be returned to the caller, whereas handle objects do not need to be returned. For more information on object behavior, see “Object Modification”.

Get Property Value

To get a property value, make a shared copy of the object as a `matlab::data::Array`. For example, assuming the object is the first input argument, assign it to a variable:

```
matlab::data::Array object(inputs[0]);
```

Create a MATLAB data array of the correct type for the property value. For example, assuming a property called `Name` contains a MATLAB character vector, define the new value as a `matlab::data::CharArray`. Use the `matlab::data::ArrayFactory` to create the array.

```
matlab::data::CharArray propName = matlabPtr->getProperty(object, u"Name");
```

Get Property Value from Object Array

If the input to the MEX function is an object array, call `getProperty` with index of the object in the array whose property value you want to get. For example, this code snippet returns the value of the `Name` property for the fourth element in the object array, `objectArray`.

```
matlab::data::Array objectArray(inputs[0]);  
matlab::data::CharArray propName = matlabPtr->getProperty(objectArray, 3, u"Name");
```

Set Property Value

To set a property value, make a shared copy of the object as a `matlab::data::Array`. For example, assuming the object is the first input argument, assign it to a variable.

```
matlab::data::Array object(inputs[0]);
```

Create a MATLAB data array of the correct type for the property value. For example, assuming a property called `Name` contains a MATLAB character vector, define the new value as a `matlab::data::CharArray`. Use the `matlab::data::ArrayFactory` to create the array.

Call the MATLAB engine `setProperty` function with the `matlabPtr` shared pointer.

```
matlabPtr->setProperty(object, u"Name",  
    factory.createCharArray("New value for Name"));
```

Set Property Value in Object Array

If the input to the MEX function is an object array, call `setProperty` with index of the object in the array whose property value you want to set. For example, this code snippet sets the value of the `Name` property for the fourth element in the object array, `objectArray`.

```
matlab::data::Array objectArray(inputs[0]);  
matlabPtr->setProperty(objectArray, 3, u"Name",  
    factory.createCharArray("New value for Name"));
```

Object Property Copy-On-Write Behavior

When a MEX function modifies an object property by assigning a value to the property, the property value is no longer shared with the object in the caller's workspace. The data in properties that are not modified remain shared. That is, copy-on-write behavior affects the modified property, not the entire object.

Modify Property and Return Object

This example uses the `EmployeeID` class to create an object. This class defines two properties. The `Name` property is defined as a 1-by-any number of elements array of type `char`. The `Picture` property is defined as a 1000-by-800 element array of type `uint8` for the employee image.

```
classdef EmployeeID
    properties
        Name (1,:) char
        Picture (1000,800) uint8
    end
    methods
        function obj = EmployeeID(n,p)
            if nargin > 0
                obj.Name = n;
                obj.Picture = p;
            end
        end
    end
end
```

The following MEX function modifies the image contained in an object of the `EmployeeID` class to reduce the intensity of the brightest values in the image. The MEX function performs these steps:

- Moves the input argument to a `matlab::data::Array` using `std::move`.
- Uses `matlab::engine::MATLABEngine::getProperty` to get the value of the property to modify.
- Defines the variable as a type appropriate for the property value. Use `matlab::data::TypedArray<uint8_t>` because the MATLAB type is `uint8`.
- Reassign the modified array to the object property using `matlab::engine::MATLABEngine::setProperty`.
- Returns the modified object as the MEX function output.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {

        // Move object to variable
        Array obj = std::move(inputs[0]);

        // Get property value and modify
        TypedArray<uint8_t> imageData = matlabPtr->getProperty(obj, u"Picture");
        for (auto& elem : imageData) {
            if (elem > 240) {
                elem = elem - elem/100;
            }
        }
    }
}
```

```

    }
    // Set property value and assign to output
    matlabPtr->setProperty(obj, u"Picture", imageData);
    outputs[0] = obj;
}
};

```

After saving this code in a file (called `reduceGlare.cpp` in this example), compile it with the `mex` function. Call this MEX function from MATLAB with an `EmployeeID` object as input.

```

EmployeeObject = EmployeeID('My Name',randi([1 255],1000,800,'uint8'));
EmployeeObject = reduceGlare(EmployeeObject);

```

Only Modified Property Causes Copy

This example creates a MEX function that accepts a `EmployeeID` object and a new value for its `Name` property as its inputs. After setting the property to the new value, the function returns the modified object. This MEX function performs some additional steps that can be useful in your program.

- Argument checking confirms the correct number of inputs and calls the MATLAB `isa` to determine the class of the input object.
- Support for either `char` or `string` input for value to set `Name` property.
- Comparison of current property value to determine if the update needs to be made.

To see the source code, click `modifyObjectProperty.cpp` and `EmployeeID.m` to open the files in the MATLAB Editor. Use the `mex` command to build the MEX function.

To run this example, add the `modifyObjectProperty.cpp` file and the `EmployeeID.m` class to a folder on your MATLAB path. After performing the `mex` setup, execute these statements:

```

mex modifyObjectProperty.cpp
EmployeeObject = EmployeeID('My Name',randi([1 255],1000,800,'uint8'));
EmployeeObject = modifyObjectProperty(EmployeeObject,'New Name');

```

Here is the `MexFunction::operator()` implementation.

```

include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {

```

```
// Check for correct inputs
checkArguments(outputs, inputs);

// Assign the input object to a matlab::data::Array
Array object(inputs[0]);

// Member function to set propertyt value
assignProperty(object, inputs);

// Return modified object
outputs[0] = object;
}
```

The `MexFunction::checkArguments` function performs these checks:

- The MEX function must be called with two arguments.
- The first argument must be an object of the `EmployeeID` class.

```
void checkArguments(ArgumentList out, ArgumentList in) {
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
    ArrayFactory factory;

    // Check number of inputs
    if (in.size() != 2) {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Two inputs required") }));
    }
    // Use isa function to test for correct class
    std::vector<Array> args{ in[0], factory.createCharArray("EmployeeID") };
    TypedArray<bool> result = matlabPtr->feval(u"isa", args);
    if (result[0] != true) {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Input must be EmployeeID object") }));
    }
}
```

The `MexFunction::assignProperty` function determines if the new value for the property is passed in as a MATLAB char vector or as a string and assigns the input defined as a `matlab::data::CharArray` or a `matlab::data::StringArray` respectively.

Before assigning the new value to the `Name` property, this function compares the current value to determine if it is different and avoids making an assignment if it is not.

```
void assignProperty(Array& obj, ArgumentList in) {
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
    ArrayFactory factory;
    std::string newPropertyValue;

    // Determine if input is MATLAB char or MATLAB string
    if (in[1].getType() == ArrayType::CHAR) {
        CharArray newName(in[1]);
        newPropertyValue = newName.toAscii();
    }
    else if (in[1].getType() == ArrayType::MATLAB_STRING) {
        StringArray newName(in[1]);
    }
}
```



```
        newPropertyValue = (std::string)newName[0];
    }
    else {
        matlabPtr->feval(u"error", 0,
            std::vector<Array>({ factory.createScalar("Name must be char or string") }));
    }

    // If new value is different from new value, set new value
    CharArray currentName = matlabPtr->getProperty(obj, u"Name");
    if (currentName.toAscii() != newPropertyValue) {
        matlabPtr->setProperty(obj, u"Name", factory.createCharArray(newPropertyValue));
    }
}
```

Handle Objects

For an example that uses a handle object, download the following file and follow the instructions in the file to build and run the MEX function.

mexgetproperty.cpp

See Also

[“matlab::engine::MATLABEngine::setProperty” on page 9-26](#) |

[“matlab::engine::MATLABEngine::getProperty” on page 9-25](#)

Related Examples

- [“Data Types for Passing MEX Function Data” on page 9-48](#)
- [“C++ MEX API” on page 9-14](#)
- [“C++ MEX Applications”](#)

Avoid Copies of Arrays in MEX Functions

MEX functions often can improve performance by controlling when the function makes copies of large data arrays. You can avoid unnecessary copies of arrays in these cases:

- The MEX function accesses an input array, but the elements of the array are not modified.
- The MEX function modifies an input array and the modified array is returned to the calling function.

Input Array Not Modified

This MEX function sums the elements of the input array, but does not modify the array. Assigning `inputs[0]` to a `const matlab::data::TypedArray<double>` variable lets you use a range-based `for` loop to sum the elements. Specifying the array as `const` ensures the variable `inArray` remains shared with the input array.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {
    ArrayFactory factory;
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        double sm = 0;
        const TypedArray<double> inArray = inputs[0];
        for (auto& elem : inArray) {
            sm += elem;
        }
        outputs[0] = factory.createScalar(sm);
    }
};
```

After saving this code in a file (called `addArrayElements.cpp` in this example), compile it with the `mex` function. Call this MEX function from MATLAB with a double array as the input argument.

```
mex addArrayElements.cpp
b = addArrayElements([1:1e7]);
```

```
b =
    5.0000e+13
```

Input Array Modified

This MEX function replaces negative values in the input array with zeros. This operation modifies the input array so the input cannot remain shared with the modified array. After validating the input array, the function does not use `inputs[0]` again. Therefore, the validated `inputs[0]` array is moved to a `matlab::data::TypedArray<double>` variable to enable the use of a range-based for loop to modify the array elements.

To prevent a copy of the input array, move `input[0]` to a `matlab::data::TypedArray<double>` using `std::move()`, which swaps the memory associated with the input array to the variable `largeArray`.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using namespace matlab::data;
using matlab::mex::ArgumentList;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        checkArguments(inputs);
        TypedArray<double> largeArray = std::move(inputs[0]);
        for (auto& elem : largeArray) {
            if (elem < 0) {
                elem = 0;
            }
        }
        outputs[0] = largeArray;
    }

    void checkArguments(ArgumentList inputs) {
        std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
        ArrayFactory factory;

        if (inputs[0].getType() != ArrayType::DOUBLE ||
            inputs[0].getType() == ArrayType::COMPLEX_DOUBLE) {
            matlabPtr->feval(u"error", 0,
                std::vector<Array>({ factory.createScalar("Incorrect input") }));
        }
    }
};
```

After saving this code in a file (called `removeNegativeNumbers.cpp` in this example), compile it with the `mex` function.

```
mex removeNegativeNumbers.cpp
```

Call this MEX function from a MATLAB function. Reassign the modified array to the same variable.

```
function array = processArray(array)
    array = removeNegativeNumbers(array);
    ....
end
```

For example, call the `processArray` function with a large array returned by the MATLAB `randn` function.

```
A = processArray(randn(10000));
min(A(:))
```

```
ans =
```

```
    0
```

See Also

Related Examples

- “Managing External Resources from MEX Functions” on page 9-34
- “Structure of C++ MEX Function” on page 9-30
- “C++ MEX Applications”

Displaying Output in MATLAB Command Window

MEX functions can display output in the MATLAB command window. However, some compilers do not support the use of `std::cout` in MEX functions. Another approach is to use `std::ostringstream` and the MATLAB `fprintf` function to display text in the MATLAB command window.

The following MEX function simply returns the text and numeric values that are passed to the function as inputs. The arguments are assumed to be a `char` and `double`. Error checking is omitted for simplicity.

Here is how the MEX function displays text in the MATLAB command window:

- Create an instance of `std::ostringstream` named `stream`.
- Insert the data to display into the `stream`.
- Call `displayOnMATLAB` with the `stream` object.

The `displayOnMATLAB` member function passes the stream contents to `fprintf` and then clears the stream buffer. You can reuse the `stream` object for subsequent calls to `displayOnMATLAB`.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::mex::ArgumentList;
using namespace matlab::data;

class MexFunction : public matlab::mex::Function {
    // Pointer to MATLAB engine to call fprintf
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();

    // Factory to create MATLAB data arrays
    ArrayFactory factory;

    // Create an output stream
    std::ostringstream stream;
public:
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        const CharArray name = inputs[0];
        const TypedArray<double> number = inputs[1];
        stream << "Here is the name/value pair that you entered." << std::endl;
        displayOnMATLAB(stream);
        stream << name.toAscii() << ": " << double(number[0]) << std::endl;
        displayOnMATLAB(stream);
    }

    void displayOnMATLAB(std::ostringstream& stream) {
        // Pass stream content to MATLAB fprintf function
        matlabPtr->feval(u"fprintf", 0,
            std::vector<Array>({ factory.createScalar(stream.str()) }));
        // Clear stream buffer
    }
};
```

```
        stream.str("");  
    };
```

Calling the MEX function (named `streamOutput.cpp` in this example) from MATLAB produces the following result.

```
mex streamOutput.cpp  
streamOutput('Total',153)
```

```
Here is the name/value pair that you entered.  
Total: 153
```

See Also

[“matlab::engine::MATLABEngine::feval” on page 9-16](#) | [matlab::data::ArrayFactory](#)

Related Examples

- [“Managing External Resources from MEX Functions” on page 9-34](#)
- [“C++ MEX API” on page 9-14](#)
- [“C++ MEX Applications”](#)

Call MATLAB from Separate Threads in MEX Function

MEX functions can call MATLAB from user-created threads using the C++ Engine asynchronous API.

Calls made asynchronously to MATLAB on a separate thread do not block MATLAB execution. The MEX function can return to the MATLAB prompt while execution on the user thread continues. You can call MATLAB functions from the user thread via the engine asynchronous API or from the command prompt. MATLAB queues the commands from each thread and executes them in the order received.

Communicating with MATLAB from Separate Threads

To call MATLAB from user-created threads, define a MEX function that uses these techniques:

- Start a thread of execution for asynchronous function calls, for example with the C++ `std::async` function.
- Use the C++ Engine API to make asynchronous calls to MATLAB.

Example to Update the Display of Text

This MEX function displays the current date and time in a MATLAB figure window. The date/time string updates every second. The MEX function returns to the MATLAB prompt while the asynchronous updates continue on the thread created by the call to `std::async`.

The `dateTimeWindow.m` function (MATLAB code) creates a figure window and a `uicontrol` text object to display the date and time. The `Tag` properties of the `uicontrol` and the figure contain identifiers used to access these objects from the MEX function.

```
function dateTimeWindow
    windowHandle = figure('MenuBar','none',...
        'ToolBar','none',...
        'Name','Current Date and Time',...
        'NumberTitle','off',...
        'Units','normalized',...
        'Position',[.01 .01 .25 .05],...
        'Tag','mexDateTimeHandle',...
        'HandleVisibility','off');
```

```
    uicontrol('Parent',windowHandle,...
             'Style','text',...
             'Tag','date_time',...
             'String',datestr(now),...
             'Units','normalized',...
             'Position',[0 0 1 1],...
             'FontSize',28);
end
```

This MEX function defines the `DisplayDateTime()` function to perform these operations:

- Calls the `dateTimeWindow.m` MATLAB function to set up the figure window and text display.
- Updates the display of the date and time once per second by assigning the output of the expression `datestr(now)` to the `uicontrol String` property.
- Tests the validity of the `uicontrol` object to determine if the figure window has been closed.
- Exits the update loop if the window and text object no longer exist.

The MEX function calls `std::async` to run the `DisplayDateTime()` member function on a separate thread.

```
/* Uses asynchronous Engine API to display date-time string
 * Calls MATLAB dateTimeWindow.m function to create figure
 * and uicontrol objects. Updates the date and time once
 * per second. Run asynchronously on a separate thread
 */

#include "mex.hpp"
#include "mexAdapter.hpp"
#include <thread>
#include <future>

class MexFunction : public matlab::mex::Function {
private:
    std::future<void> voidStdFuture;
    std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
    bool isRunning = false;
public:

    void DisplayDateTime() {
        matlab::data::ArrayFactory factory;
        matlabPtr->evalAsync(u"dateTimeWindow;");
        while (isRunning) {
            matlabPtr->evalAsync(u"set(findall(0,'Tag','date_time'),
                'String',datestr(now));");
            std::vector<matlab::data::Array> args({
                factory.createScalar<double>(0),
                factory.createCharArray("Tag"),
                factory.createCharArray("date_time"),
            });
            voidStdFuture = std::async(std::launch::async,
                [matlabPtr, args] {
                    matlabPtr->evalAsync(u"set(findall(0,'Tag','date_time'),
                        'String',datestr(now));");
                });
            voidStdFuture.wait();
        }
    }
};
```



```

    });
    matlab::engine::FutureResult<matlab::data::Array> fresult;
    fresult = matlabPtr->fevalAsync(u"findall", args);
    matlab::data::Array result = fresult.get();
    isRunning = !result.isEmpty();
    if (!isRunning) { matlabPtr->evalAsync(u"mexDateTime close"); }
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
}

void operator()(matlab::mex::ArgumentList outputs,
               matlab::mex::ArgumentList inputs) {
    if (inputs.size() == 0) {
        mexLock();
        if(!isRunning) {
            isRunning = true;
            voidStdFuture = std::async(std::launch::async,
                                     &MexFunction::DisplayDateTime, this);
        }
    }
    else {
        isRunning = false;
        matlabPtr->eval(u"close(findall(0,'Tag','mexDateTimeHandle'))");
        mexUnlock();
    }
}
};

```

To use the MEX function saved as `mexDateTime.cpp`, use the `mex` command to build the program.

```

mex -setup c++
mex mexDateTime.cpp
mexDateTime

```

The MEX function locks the MEX file to prevent attempts to recompile the MEX function while the separate thread is still active. The MEX function unlocks itself when you end execution.

To end execution on the separate thread, close the figure window containing the date and time text or call the MEX function with an input argument. For example:

```
mexDateTime close
```

See Also

`figure` | `matlab::mex::Function` | `uicontrol`

Related Examples

- “C++ MEX API” on page 9-14
- “Call MATLAB Functions from MEX Functions” on page 9-52

Out-of-Process Execution of C++ MEX Functions

MATLAB can run C++ MEX functions in a separate process. Running a C++ MEX function in a separate process enables you to:

- Isolate the MATLAB process from crashes in the C++ MEX function.
- Use some third-party libraries in the C++ MEX function that are not compatible with MATLAB.
- Conserve memory by running multiple C++ MEX functions in a single process.
- Attach C++ debuggers to the MEX host process.
- Interrupt C++ MEX function execution using **Ctrl+C**

Note The size of variables passed between C++ and MATLAB is limited to 2 GB when you call a C++ function out-of-process. This limit applies to the data plus supporting information passed between the processes.

How to Run Out of Process

Follow these steps to run your C++ MEX function out of process:

- 1 Write your C++ MEX function and build it using the instructions in “Build C++ MEX Programs” on page 9-10. There are no required code differences between functions written for in-process and out-of-process execution.
- 2 Create a MEX host process using the `mexhost` function.
- 3 Use the `feval` method of the `matlab.mex.MexHost` object returned by `mexhost` to run your C++ MEX function in the host process.

When running out of process, C++ MEX functions always execute in the same folder as the MATLAB current folder. Changing the MATLAB current folder after creating the MEX host object results in the same change in the C++ MEX function context.

Running `arrayProduct` MEX Function Out of Process

The following example runs the `arrayProduct` C++ MEX function out-of-process. The C++ MEX source code is available in the file `arrayProduct.cpp`. To use this example C++ MEX function, open the `arrayProduct.cpp` source file, save it on your MATLAB path,

and build the C++ MEX function using the instructions in “Build C++ MEX Programs” on page 9-10.

After building the C++ MEX function, start the MEX host process using the `mexhost` function. This function returns an object of the `matlab.mex.MexHost` class.

Use the `feval` method to run the `arrayProduct` C++ MEX function in the host process. Pass the C++ MEX functions name and arguments to `feval`. Return the results to MATLAB by assigning the output of `feval`.

```
mh = mexhost;
result = feval(mh, 'arrayProduct', 2, [1 2 3 4])

result =
     2     4     6     8
```

Run `arrayProduct` in another MEX host process. First create the MEX host process by calling `mexhost` again and assigning the output to a different variable. Then call `feval` on the new host.

```
mh2 = mexhost;
result2 = feval(mh2, 'arrayProduct', 2, rand(1,10));
```

You can run other C++ MEX functions in the same process using the same `matlab.mex.MexHost` object returned by `mexhost`.

```
result2 = feval(mh2, 'anotherMexFunction', inputs);
```

Process Life Cycle

The MEX host process life cycle is coupled to the life cycle of the `matlab.mex.MexHost` object returned by `mexhost`. Like any handle object, MATLAB invokes the `delete` method when the object is no longer referenced anywhere. If MATLAB invokes `delete` or if you call `delete` on the object explicitly, MATLAB ends the process associated with the `MexHost` object.

You can end the process explicitly by calling the `clear` function with any of these options:

- `clear` the variable returned by the `mexhost` function
- `clear classes`, `clear java`, or `clear all`

Rebuilding a C++ MEX Function

The `mex` command automatically unloads C++ MEX functions before rebuilding them. Therefore, you do not need to explicitly unload C++ MEX functions from the MEX host process.

You can explicitly unload all C++ MEX functions from a MEX host process by calling `clear mex` or `clear functions`. Unload a specific C++ MEX function by calling `clear` on the function name.

To prevent the unloading of a C++ MEX function, use the `mexLock` function. To unlock the C++ MEX function, use `mexUnlock`. These functions set or unset a flag for the C++ MEX function on the host process to control the unloading of C++ MEX functions.

Getting Information About the MEX Host Process

Use the `matlab.mex.MexHost` object to get information about the MEX host process. These properties provide information about the process and loaded functions.

- `EnvironmentVariables` contains names and values of environment variables set for the process.
- `Functions` contains the names of all C++ MEX functions loaded into the MEX host process.
- `ProcessName` contains the name of the MEX host process, which is `MATLABMexHost` by default.
- `ProcessIdentifier` contains the process identifier.

For more information on these properties, see `matlab.mex.MexHost`.

Always Run Out of Process

In certain cases, you might always want to run your C++ MEX function out-of-process. For example, running out of process can be necessary if there are third-party library conflicts between MATLAB and the C++ MEX function.

To provide a convenient way to run the C++ MEX function out of process, create a wrapper MATLAB function. The wrapper function creates the MEX host object and runs the C++ MEX function in that process.

For example, suppose that you want to create a wrapper function for the `arrayProduct.cpp` C++ MEX function that is included in the documentation as an

example. Create a MATLAB function with the name `arrayProduct.m` and put this file in a folder that is on the MATLAB path. Build the C++ MEX function and assign a different name for the compiled MEX file.

The wrapper function creates a `matlab.mex.MexHost` object that is assigned to a persistent variable so that the process is not destroyed between subsequent calls to the wrapper function. The function uses this object to call the `feval` method for out-of-process execution of the C++ MEX function. If the MEX host object is not valid, then the function creates a MEX host process.

```
function result = arrayProduct(scalefactor,inputarray)
    persistent mh
    if ~(isa(mh,'matlab.mex.MexHost') && isvalid(mh))
        mh = mexhost;
    end
    result = feval(mh,"arrayProductMEX",scalefactor,inputarray);
end
```

To build the C++ MEX function, open the `arrayProduct.cpp` source file and save it on your MATLAB path. Build the C++ MEX function using the instructions in this topic, “Build C++ MEX Programs” on page 9-10.

The following command builds MEX file with the root name `arrayProductMEX` and puts it in the folder with the wrapper function, which is assumed to be `MyPathFolder` in this example. The `mex` command creates the folder if it does not exist.

```
mex -output arrayProductMEX -outdir MyPathFolder arrayProduct.cpp
```

To use the C++ MEX function, call it from the command line like any function via the wrapper function.

```
result = arrayProduct(2,[1 2 3 4]);
result

result =

     2     4     6     8
```

The following code calls the C++ MEX function in a `for` loop. The first call to `arrayProduct` creates the MEX host process. Subsequent calls use the same process unless the process is destroyed by, for example, a crash of the C++ MEX function.

```
for k = 1:5
    results(k,:) = arrayProduct(k,[1 2 3 4]);
end
```

```
end
results
results =
     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16
     5    10    15    20
```

To destroy the MEX host process, clear all functions that define the MEX host process variable (`mh` in this example) as persistent. In this example, clear the `arrayProduct.m` function.

```
clear arrayProduct.m
```

Calling `clear functions` causes the destruction of the `matlab.mex.MexHost` object that is stored in the persistent variable and therefore terminates the MEX host process.

Debugging Out-of-Process MEX Functions

You can attach a C++ debugger to your C++ MEX function and set break points to debug your program. Here are the steps to set up and debug a C++ MEX function out of process.

- Build C++ MEX source code using the `mex` command with the `-g` option to include debugging symbols.
- Create a host process using the `mexhost` function. This function returns an object of the `matlab.mex.MexHost`.
- Get the process identifier from the `matlab.mex.MexHost` object `ProcessIdentifier` property.
- Use the process identifier to attach a C++ debugger to the process.
- Insert breakpoints in the source code.
- Run the C++ MEX function out-of-process using the `feval` method.

For information on using specific debuggers, see the documentation for those debuggers.

Debugging Using Microsoft Visual Studio

- 1 Ensure Visual Studio is your selected C++ compiler. This example uses Microsoft Visual Studio 2015.

```
cpp = mex.getCompilerConfigurations('C++', 'Selected');  
cpp.Name  
  
ans =  
  
    'Microsoft Visual C++ 2015'
```

- 2 Build your C++ MEX source code using the `mex` command with the `-g` option. This example assumes that you have a C++ MEX source file named `myMexFunction.cpp`.

```
mex -g myMexFunction.cpp
```

- 3 Create a MEX host process and return the `MexHost` object.

```
mh = mexhost;
```

- 4 Start Visual Studio. Do not exit your MATLAB session.
- 5 From the Visual Studio **Debug** menu, select **Attach to Process**.

In the Attach to Process dialog box, select the `MATLABMexHost` process, and click **Attach**.

- 6 Visual Studio loads data, and then displays an empty code pane.
- 7 Open your C++ MEX source file by clicking **File > Open > File** and selecting the file.
- 8 Set a breakpoint by right-clicking the desired line of code and clicking **Breakpoint > Insert Breakpoint** on the context menu.
- 9 In MATLAB, run the C++ MEX function out of process using the `matlab.mex.MexHost feval` method.

```
result = feval(mh, 'myMexFunction', input1, input2, ...)
```

- 10 Use the features provided by the debugger to debug your source code.

Debugging on Linux Systems

On Linux systems, you can use a debugger such as the GNU `gdb` debugger. Follow these steps to use the `gdb` debugger.

- 1 Build your C++ MEX source code using the `mex` command with the `-g` option. This example assumes that you have a C++ MEX source file named `myMexFunction.cpp`.

```
mex -g myMexFunction.cpp
```


- 2 Create a MEX host process and return the `MexHost` object.

```
mh = mexhost;
```

- 3 Get the process identifier from the `ProcessIdentifier` property of the `MexHost` object. The returned value is a string representing the identifier of the MEX host process. For example,

```
mh.ProcessIdentifier
```

```
ans =
```

```
"13892"
```

The process identifier is different for each process created.

- 4 Attach a debugger to the MEX host process from the Linux terminal. For example, using the GNU `gdb` debugger, call `gdb` with the C++ MEX file name and the process identifier that you obtained from the MEX host object in MATLAB:

```
gdb myMexFunction -pid=13892
```

- 5 Set break points in the C++ MEX function. For example, using `gdb`, set the break point in `myMexFunction.cpp` at line 21:

```
break myMexFunction.cpp:21
```

- 6 From MATLAB, run the C++ MEX function out-of-process using the `matlab.mex.MexHost feval` method.

```
result = feval(mh, 'myMexFunction', input1, input2, ...)
```

MATLAB waits for a response from the debugger.

- 7 From the Linux terminal, use the features provided by the debugger to debug your source code.

Debugging on Macintosh Systems

On Macintosh systems, use the LLDB debugger.

- 1 Build your C++ MEX source code using the `mex` command with the `-g` option. This example assumes that you have a C++ MEX source file named `myMexFunction.cpp`.

```
mex -g myMexFunction.cpp
```

- 2 Create a MEX host process and return the `MexHost` object.

```
mh = mexhost;
```

- 3 Get the process identifier from the `ProcessIdentifier` property of the `MexHost` object. The returned value is a string representing the identifier of the MEX host process.

```
mh.ProcessIdentifier
```

```
ans =
```

```
"13892"
```

The process identifier is different for each process created.

- 4 Attach a debugger to the MEX host process from the macOS Terminal. Call `lldb` with the MEX host process identifier that you obtained from the MEX host object in MATLAB. For example, assuming the process identifier is 13892, attach the LLDB debugger to this process:

```
lldb -p 13892
```

- 5 Set a break points in your source code. For example, this command sets a break point in `myMexFunction.cpp` at line number 21.

```
breakpoint set -f myMexFunction.cpp -l 21
```

- 6 From MATLAB, run the C++ MEX function out-of-process using the `matlab.mex.MexHost feval` method.

```
result = feval(mh'myMexFunction',input1,input2,...)
```

MATLAB waits for a response from the debugger.

- 7 **Enter C** or **continue** from the macOS Terminal. Program execution stops at breakpoints.
- 8 From the macOS Terminal, use the features provided by the debugger to debug your source code.

See Also

`matlab.mex.MexHost` | `mexhost`

Related Examples

- “Call MATLAB from Separate Threads in MEX Function” on page 9-71
- “C++ MEX Applications”

Making async Requests Using `mexCallMATLAB`

Be aware of the behavior when calling back into MATLAB using `mexCallMATLAB` with an async request like `matlab::engine::MATLABEngine::fevalAsync`. The `mexCallMATLAB` function is synchronous. After it launches a function using the call to `fevalAsync`, the MEX function continues to the end, returns to MATLAB, and is then unloaded. If the async processing of the requested `feval` command tries to reference the MEX function, it generates an exception.

Consider using the `mexLock` and `mexUnlock` functions to ensure that the MEX function stays in memory while the async function processes.

See Also

More About

- “C++ MEX API” on page 9-14

Fortran MEX-Files

- “Components of Fortran MEX File” on page 10-2
- “MATLAB Fortran API Libraries” on page 10-5
- “Data Flow in Fortran MEX Files” on page 10-8
- “User Messages” on page 10-11
- “Error Handling” on page 10-12
- “Build Fortran MEX File” on page 10-13
- “Create Fortran Source MEX File” on page 10-14
- “Set Up Fortran Examples” on page 10-19
- “Dynamically Allocate Memory” on page 10-20
- “Call MATLAB Functions from Fortran MEX Files” on page 10-21
- “Debug Fortran MEX Files” on page 10-23
- “Handling Large mxArray’s” on page 10-26
- “Memory Management” on page 10-29
- “MATLAB Supports Fortran 77” on page 10-30
- “Handle Complex Fortran Data” on page 10-31

Components of Fortran MEX File

In this section...
"mexFunction Gateway Routine" on page 10-2
"Naming the MEX File" on page 10-2
"Difference Between .f and .F Files" on page 10-2
"Required Parameters" on page 10-3
"Managing Input and Output Parameters" on page 10-3
"Validating Inputs" on page 10-4
"Computational Routine" on page 10-4

mexFunction Gateway Routine

The gateway routine is the entry point to the MEX file. It is through this routine that MATLAB accesses the rest of the routines in your MEX files. The name of the gateway routine is `mexFunction`. It takes the place of the main program in your source code.

Naming the MEX File

The name of the source file containing `mexFunction` is the name of your MEX file, and, hence, the name of the function you call in MATLAB. Name your Fortran source file with an uppercase `.F` file extension.

The file extension of the binary MEX file is platform-dependent. You find the file extension using the `mexext` function, which returns the value for the current machine.

Difference Between .f and .F Files

To ensure that your Fortran MEX file is platform independent, use an uppercase `.F` file extension.

Fortran compilers assume source files using a lowercase `.f` file extension have been preprocessed. On most platforms, `mex` makes sure that the file is preprocessed regardless of the file extension. However, on Apple Macintosh platforms, `mex` cannot force preprocessing.

Required Parameters

The Fortran signature for `mexfunction` is:

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

Place this subroutine after your computational routine and any other subroutines in your source file.

The following table describes the parameters for `mexFunction`.

Parameter	Description
<code>prhs</code>	Array of right-side input arguments.
<code>plhs</code>	Array of left-side output arguments.
<code>nrhs</code>	Number of right-side arguments, or the size of the <code>prhs</code> array.
<code>nlhs</code>	Number of left-side arguments, or the size of the <code>plhs</code> array.

Declare `prhs` and `plhs` as type `mwPointer`, which means they point to MATLAB arrays. They are vectors that contain pointers to the arguments of the MEX file.

You can think of the name `prhs` as representing the “parameters, right-hand side,” that is, the input parameters. Likewise, `plhs` represents the “parameters, left side,” or output parameters.

Managing Input and Output Parameters

Input parameters (found in the `prhs` array) are read-only; do not modify them in your MEX file. Changing data in an input parameter can produce undesired side effects.

You also must take care when using an input parameter to create output data or any data used locally in your MEX file. If you want to copy an input array into an output array, for example `plhs(1)`, call the `mxDuplicateArray` function to make a copy of the input array. For example:

```
plhs(1) = mxDuplicateArray(prhs(1))
```

For more information, see the troubleshooting topic “Incorrectly Constructing a Cell or Structure `mxArray`” on page 7-87.

Validating Inputs

For a list of functions to validate inputs to your subroutines, see the Matrix Library category, “Validate Data”. The `mxIsClass` function is a general-purpose way to test an `mxArray`.

Computational Routine

The computational routine contains the code for performing the computations you want implemented in the binary MEX file. Although not required, consider writing the gateway routine, `mexFunction`, to call a computational routine. To validate input parameters and to convert them into the types required by the computational routine, use the `mexFunction` code as a wrapper.

If you write separate gateway and computational routines, you can combine them into one source file or into separate files. If you use separate files, the file containing `mexFunction` must be the first source file listed in the `mex` command.

See Also

`mexFunction` | `mexext` | `mwPointer` | `mxDuplicateArray` | `mxIsClass`

MATLAB Fortran API Libraries

In this section...

“Matrix Library” on page 10-5

“MEX Library” on page 10-5

“Preprocessor Macros” on page 10-5

“Using the Fortran %val Construct” on page 10-6

The “Fortran Matrix API” and the “Fortran MEX API” describe functions you can use in your gateway and computational routines that interact with MATLAB programs and the data in the MATLAB workspace.

To use these functions, include the `fintrf` header, which declares the entry point and interface routines. The file is in your `matlabroot\extern\include` folder. Put this statement in your source file:

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

Matrix Library

Use Fortran Matrix API functions to pass `mxArray`, the type MATLAB uses to store arrays, to and from MEX files. For examples using these functions, see `matlabroot/extern/examples/mx`.

MEX Library

Use Fortran MEX API functions to perform operations in the MATLAB environment. For examples using these functions, see `matlabroot/extern/examples/mex`.

Unlike MATLAB functions, MEX file functions do not have their own variable workspace. MEX file functions operate in the caller workspace. To evaluate a string, use `mexEvalString`. To get and put variables into the caller workspace, use the `mexGetVariable` and `mexPutVariable` functions.

Preprocessor Macros

The Matrix and MEX libraries use the MATLAB preprocessor macros `mwSize` and `mwIndex` for cross-platform flexibility. `mwSize` represents size values, such as array

dimensions and number of elements. `mwIndex` represents index values, such as indices into arrays.

MATLAB has an extra preprocessor macro for Fortran files, `mwPointer`. MATLAB uses a unique data type, the `mxAArray`. Because you cannot create a data type in Fortran, MATLAB passes a special identifier, created by the `mwPointer` preprocessor macro, to a Fortran program. This is how you get information about an `mxAArray` in a native Fortran data type. For example, you can find out the size of the `mxAArray`, determine whether it is a string, and look at the contents of the array. Use `mwPointer` to build platform-independent code.

The Fortran preprocessor converts `mwPointer` to `integer*4` when building binary MEX files on 32-bit platforms and to `integer*8` when building on 64-bit platforms.

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

Using the Fortran `%val` Construct

The Fortran `%val(arg)` construct specifies that an argument, *arg*, is to be passed by value, instead of by reference. Most, but not all, Fortran compilers support the `%val` construct.

If your compiler does not support the `%val` construct, copy the array values into a temporary true Fortran array using the `mxCopy*` routines (for example, `mxCopyPtrToReal8`).

%val Construct Example

If your compiler supports the `%val` construct, you can use routines that point directly to the data (that is, the pointer returned by typed data access functions like `mxDoubles` or `mxDoublesComplex`). You can use `%val` to pass the contents of this pointer to a subroutine, where it is declared as a Fortran double-precision matrix.

For example, consider a gateway routine that calls its computational routine, `yprime`, by:

```
call yprime(%val(y), %val(t), %val(y))
```

If your Fortran compiler does not support the `%val` construct, you would replace the call to the computational subroutine with:

```
C Copy array pointers to local arrays.  
    call mxCopyPtrToReal8(t, tr, 1)  
    call mxCopyPtrToReal8(y, yr, 4)  
C  
C Call the computational subroutine.  
    call yprime(ypr, tr, yr)  
C  
C Copy local array to output array pointer.  
    call mxCopyReal8ToPtr(ypr, yp, 4)
```

You must also add the following declaration line to the top of the gateway routine:

```
real*8 ypr(4), tr, yr(4)
```

If you use `mxCopyPtrToReal8` or any of the other `mxCopy*` routines, the size of the arrays declared in the Fortran gateway routine must be greater than or equal to the size of the inputs to the MEX file coming in from MATLAB. Otherwise, `mxCopyPtrToReal8` does not work correctly.

See Also

[“Fortran Matrix API”](#) | [“Fortran MEX API”](#) | [mwIndex](#) | [mwPointer](#) | [mwSize](#) | [mxArray](#)

Data Flow in Fortran MEX Files

In this section...

“Showing Data Input and Output” on page 10-8

“Gateway Routine Data Flow Diagram” on page 10-9

Showing Data Input and Output

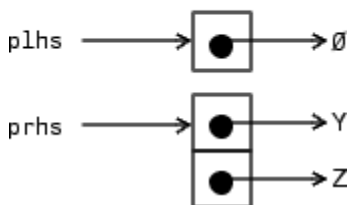
Suppose your MEX-file `myFunction` has two input arguments and one output argument. The MATLAB syntax is `[X] = myFunction(Y, Z)`. To call `myFunction` from MATLAB, type:

```
X = myFunction(Y, Z);
```

The MATLAB interpreter calls `mexFunction`, the gateway routine to `myFunction`, with the following arguments:

```
nlhs = 1
```

```
nrhs = 2
```



Your input is `prhs`, a two-element array (`nrhs = 2`). The first element is a pointer to an `mxArray` named `Y` and the second element is a pointer to an `mxArray` named `Z`.

Your output is `plhs`, a one-element array (`nlhs = 1`) where the single element is a null pointer. The parameter `plhs` points at nothing because the output `X` is not created until the subroutine executes.

The gateway routine creates the output array and sets a pointer to it in `plhs[0]`. If the routine does not assign a value to `plhs[0]` but you assign an output value to the function when you call it, MATLAB generates an error.

Note It is possible to return an output value even if $n\text{lhs} = 0$, which corresponds to returning the result in the `ans` variable.

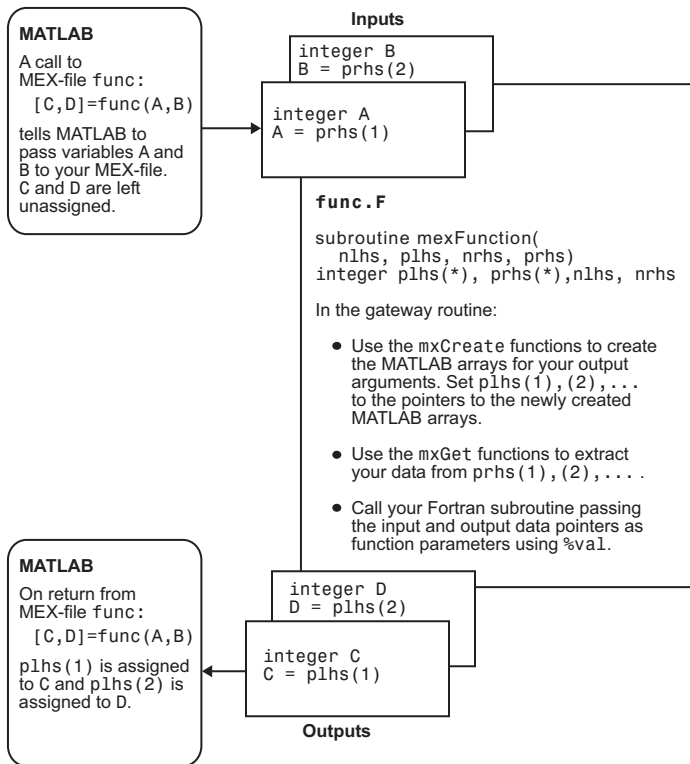
Gateway Routine Data Flow Diagram

The following MEX Cycle diagram shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.

In this example, the syntax of the MEX-file `func` is `[C, D] = func(A,B)`. In the figure, a call to `func` tells MATLAB to pass variables `A` and `B` to your MEX-file. `C` and `D` are left unassigned.

The gateway routine uses the `mxCreate*` functions to create the MATLAB arrays for your output arguments. It sets `plhs[0]` and `plhs[1]` to the pointers to the newly created MATLAB arrays. It uses the `mxGet*` functions to extract your data from your input arguments `prhs[0]` and `prhs[1]`. Finally, it calls your computational routine, passing the input and output data pointers as function parameters.

MATLAB assigns `plhs[0]` to `C` and `plhs[1]` to `D`.



Fortran MEX Cycle

User Messages

To print text in the MATLAB Command Window, use the `mexPrintf` function. To print error and warning information in the Command Window, use the `mexErrMsgIdAndTxt` and `mexWarnMsgIdAndTxt` functions.

For example, the following code snippet displays an error message.

```
parameter(maxbuf = 100)
character*100 input_buf

if (status = mxGetString(prhs(1), input_buf, maxbuf) .ne. 0) then
    call mexErrMsgIdAndTxt ('MATLAB:myfunc:readError',
+                          'Error reading string.')
endif
```

See Also

[mexErrMsgIdAndTxt](#) | [mexPrintf](#) | [mexWarnMsgIdAndTxt](#)

Error Handling

The `mexErrMsgIdAndTxt` function prints error information and terminates your binary MEX file. For an example, see the following code in `matlabroot/extern/examples/mx/mxcreatecellmatrixf.F`.

```
C      Check for proper number of input and output arguments
      if (nrhs .lt. 1) then
          call mexErrMsgIdAndTxt( 'MATLAB:mxcreatecellmatrixf:minrhs',
+   'One input argument required.')
      end if
```

The `mexWarnMsgIdAndTxt` function prints information, but does not terminate the MEX file. For an example, see the following code in `matlabroot/extern/examples/mx/mxgetepsf.F`.

```
C      Check for equality within eps
      do 20 j=1,elements
          if ((abs(first(j) - second(j)).gt.(abs(second(j)*eps))) then
              call mexWarnMsgIdAndTxt(
+   'MATLAB:mxgetepsf:NotEqual',
+   'Inputs are not the same within eps.')
              go to 21
          end if
      20 continue
```

See Also

`mexErrMsgIdAndTxt` | `mexWarnMsgIdAndTxt`

Related Examples

- `mxcreatecellmatrixf.F`
- `mxgetepsf.F`

Build Fortran MEX File

This example shows how to build the example MEX file, `timestwo`. Use this example to verify the build configuration for your system.

To build a code example, first copy the file to a writable folder, such as `c:\work`, on your path:

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'refbook', 'timestwo.F'), '.', 'f')
```

Use the `mex` command to build the MEX file.

```
mex timestwo.F
```

This command creates the file `timestwo.ext`, where `ext` is the value returned by the `mexext` function.

The `timestwo` function takes a scalar input and doubles it. Call `timestwo` as if it were a MATLAB function.

```
timestwo(4)
```

```
ans =  
     8
```

See Also

[mex](#) | [mexext](#)

More About

- “Handling Large mxArray in C MEX Files” on page 8-62
- “Upgrade MEX Files to Use 64-Bit API” on page 7-51

Create Fortran Source MEX File

This example shows how to write a MEX file to call a Fortran subroutine, `timestwo`, in MATLAB using a MATLAB matrix. You can view the complete source file here on page 10-0 . This example uses the MATLAB Editor to write the source code and the MATLAB `mex` command to create the MEX function.

Fortran subroutine `timestwo`

The following code defines the `timestwo` subroutine, which multiplies an n-dimensional array, `x_input`, by 2, and returns the results in array, `y_output`.

```
subroutine timestwo(y_output, x_input)
  real*8 x_input, y_output

  y_output = 2.0 * x_input
  return
end
```

Create Source File

Open MATLAB Editor, create a file, and document the MEX file with the following information.

```
C=====
C   timestwo.f
C   Computational function that takes a scalar and doubles it.
C   This is a MEX file for MATLAB.
C=====
```

Add the Fortran header file, `fintrf.h`, containing the MATLAB API function declarations.

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

Save the file on your MATLAB path, for example, in `c:\work`, and name it `timestwo.F`. The name of your MEX file is `timestwo`.

Create Gateway Routine

MATLAB uses the gateway routine, `mexfunction`, as the entry point to a Fortran subroutine. Add the following `mexFunction` code.

```
C   Gateway routine
C   subroutine mexFunction(nlhs, plhs, nrhs, prhs)
```

```
C    Declarations
```

```
C    Statements
```

```
    return
    end
```

Add the following statement to your `mexfunction` subroutine to force you to declare all variables.

```
implicit none
```

Explicit type declaration is necessary for 64-bit arrays.

Declare `mexfunction` Arguments

To declare `mxArray` variables, use the MATLAB type, `mwPointer`. Add this code after the `Declarations` statement.

```
C    mexFunction arguments:
    mwPointer plhs(*), prhs(*)
    integer nlhs, nrhs
```

Declare Functions and Local Variables

- Declare the symbolic names and types of MATLAB API functions used in this MEX file.

```
C    Function declarations:
    mwPointer mxGetDoubles
    mwPointer mxCreateDoubleMatrix
    integer mxIsNumeric
    mwPointer mxGetM, mxGetN
```

To determine the type of a function, refer to the MATLAB API function reference documentation. For example, see the documentation for `mxGetDoubles`.

- Declare local variables for the `mexfunction` arguments.

```
C    Pointers to input/output mxArray:
    mwPointer x_ptr, y_ptr
```

- Declare matrix variables.

```
C    Array information:
    mwPointer mrows, ncols
    mwSize size
```

Verify MEX File Input and Output Arguments

Verify the number of MEX file input and output arguments using the `nrhs` and `nlhs` arguments. Add these statements to the `mexfunction` code block.

```
C    Check for proper number of arguments.
    if(nrhs .ne. 1) then
        call mexErrMsgIdAndTxt ('MATLAB:timestwo:nInput',
+                               'One input required.')
    elseif(nlhs .gt. 1) then
        call mexErrMsgIdAndTxt ('MATLAB:timestwo:nOutput',
+                               'Too many output arguments.')
    endif
```

Verify the input argument type using the `prhs` argument.

```
C    Check that the input is a number.
    if(mxIsNumeric(prhs(1)) .eq. 0) then
        call mexErrMsgIdAndTxt ('MATLAB:timestwo:NonNumeric',
+                               'Input must be a number.')
    endif
```

Create Computational Routine

Add the `timestwo` code. This subroutine is your computational routine, the source code that performs the functionality you want to use in MATLAB.

```
C    Computational routine

    subroutine timestwo(y_output, x_input)
    real*8 x_input, y_output

    y_output = 2.0 * x_input
    return
    end
```

A computational routine is optional. Alternatively, you can place the code within the `mexfunction` function block.

Declare Variables for Computational Routine

Put the following variable declarations in `mexFunction`.

```
C    Arguments for computational routine:
    real*8 x_input, y_output
```

Read Input Array

To point to the input matrix data, use the `mxGetDoubles` function.

```
x_ptr = mxGetDoubles(prhs(1))
```

To create a Fortran array, `x_input`, use the `mxCopyPtrToReal8` function.

```

C    Get the size of the input array.
    mrows = mxGetM(prhs(1))
    ncols = mxGetN(prhs(1))
    size = mrows*ncols

C    Create Fortran array from the input argument.
    call mxCopyPtrToReal8(x_ptr,x_input,size)

```

Prepare Output Data

To create the output argument, `plhs(1)`, use the `mxCreateDoubleMatrix` function.

```

C    Create matrix for the return argument.
    plhs(1) = mxCreateDoubleMatrix(mrows,ncols,0)

```

Use the `mxGetDoubles` function to assign the `y_ptr` argument to `plhs(1)`.

```
y_ptr = mxGetDoubles(plhs(1))
```

Perform Calculation

Pass the arguments to `timestwo`.

```

C    Call the computational subroutine.
    call timestwo(y_output, x_input)

```

Copy Results to Output Argument

```

C    Load the data into y_ptr, which is the output to MATLAB.
    call mxCopyReal8ToPtr(y_output,y_ptr,size)

```

View Complete Source File

Compare your source file with `timestwo.F`, located in the `matlabroot/extern/examples/refbook` folder. Open the file in the editor.

Build Binary MEX File

At the MATLAB command prompt, build the binary MEX file.

```
mex -R2018a timestwo.F
```

Test the MEX File

```
x = 99;  
y = timestwo(x)  
  
y =  
    198
```

See Also

[mexfunction](#) | [mwPointer](#) | [mwSize](#) | [mxCreateDoubleMatrix](#) | [mxGetDoubles](#) | [mxGetM](#) | [mxGetN](#) | [mxIsNumeric](#)

Related Examples

- [timestwo.F](#)

Set Up Fortran Examples

The “Fortran Matrix API” provides a set of Fortran routines that handle the types supported by MATLAB. For each data type, there is a specific set of functions that you can use for data manipulation.

Source code for Fortran examples is located in the *matlabroot/extern/examples/refbook* folder of your MATLAB installation. To build an example, first copy the file to a writable folder, such as *c:\work*, on your path:

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','filename.F'),'.','f')
```

where *filename* is the name of the example.

Make sure that you have a Fortran compiler installed. Then, at the MATLAB command prompt, type:

```
mex filename.F
```

where *filename* is the name of the example.

This section looks at source code for the examples. Unless otherwise specified, the term “MEX file” refers to a source file.

For a list of MEX example files available with MATLAB, see “Tables of MEX Function Source Code Examples” on page 8-20.

Dynamically Allocate Memory

To allocate memory dynamically in a Fortran MEX file, use `%val`. (See “Using the Fortran `%val` Construct” on page 10-6.) The example `dblmat.F` takes an input matrix of real data and doubles each of its elements. To see the source code, open the file in MATLAB Editor. `compute.F` is the subroutine `dblmat` calls to double the input matrix. (Open the file in MATLAB Editor.)

To build this example, at the command prompt type:

```
mex dblmat.F compute.F
```

For the 2-by-3 matrix, type:

```
x = [1 2 3; 4 5 6];  
y = dblmat(x)
```

```
y =  
    2     4     6  
    8    10    12
```

Note The `dblmat.F` example, as well as `fulltosparse.F` and `sincall.F`, are split into two parts, the gateway and the computational subroutine, because of restrictions in some compilers.

Call MATLAB Functions from Fortran MEX Files

You can call MATLAB functions, operators, user-defined functions, and other binary MEX files from within your Fortran source code by using the API function `mexCallMATLAB`. The `sincall.F` example creates an `mxArray`, passes various pointers to a local function to acquire data, and calls `mexCallMATLAB` to calculate the sine function and plot the results. To see the source code, open the file in MATLAB Editor. `fill.F` is the subroutine `sincall` calls to fill the `mxArray` with data. (Open the file in MATLAB Editor.)

It is possible to use `mexCallMATLAB` (or any other API routine) from within your computational Fortran subroutine. You can only call most MATLAB functions with double-precision data. Some functions that perform computations, such as the `eig` function, do not work correctly with data that is not double precision.

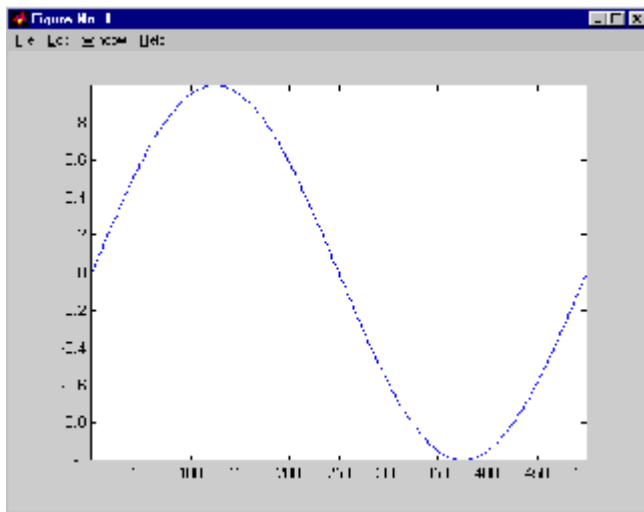
To build this example, at the command prompt type:

```
mex -R2018a sincall.F fill.F
```

Running this example:

```
sincall
```

displays the results:



Note You can generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. See the following 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 of type `mxUNKNOWN_CLASS`.

Debug Fortran MEX Files

In this section...

“Notes on Debugging” on page 10-23

“Debugging on Microsoft Windows Platforms” on page 10-23

“Debugging on Linux Platforms” on page 10-23

Notes on Debugging

The examples show how to debug `timestwo.F`, found in your `matlabroot/extern/examples/refbook` folder.

Binary MEX files built with the `-g` option do not execute on other computers because they rely on files that are not distributed with MATLAB software. For more information on isolating problems with MEX files, see Troubleshooting on “C MEX File Applications”.

Debugging on Microsoft Windows Platforms

For MEX files compiled with any version of the Intel® Visual Fortran compiler, you can use the debugging tools found in your version of Microsoft Visual Studio.

Debugging on Linux Platforms

The MATLAB supported Fortran compiler `g95` has a `-g` option for building binary MEX files with debug information. Such files can be used with `gdb`, the GNU Debugger. This section describes using `gdb`.

GNU Debugger `gdb`

In this example, the MATLAB command prompt `>>` is shown in front of MATLAB commands, and `linux>` represents a Linux prompt; your system might show a different prompt. The debugger prompt is `<gdb>`.

- 1 To compile the source MEX file, type:

```
linux> mex -g timestwo.F
```

- 2 At the Linux prompt, start the `gdb` debugger using the `matlab -D` option:

```
linux> matlab -Dgdb
```

- 3 Start MATLAB without the Java Virtual Machine (JVM) by using the `-nojvm` startup flag:

```
<gdb> run -nojvm
```

- 4 In MATLAB, enable debugging with the `dbmex` function and run your binary MEX file:

```
>> dbmex on  
>> y = timestwo(4)
```

- 5 You are ready to start debugging.

It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

Note The compiler might alter the function name. For example, it might append an underscore. To determine how this symbol appears in a given MEX file, use the Linux command `nm`. For example:

```
linux> nm timestwo.mexa64 | grep -i mexfunction
```

The operating system responds with something like:

```
0000091c T mexfunction_
```

Use `mexFunction` in the breakpoint statement. Be sure to use the correct case.

```
<gdb> break mexfunction_  
<gdb> continue
```

- 6 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type `continue`:

```
<gdb> continue
```

- 7 After stopping at the last breakpoint, type:

```
<gdb> continue
```

`timestwo` finishes and MATLAB displays:

y =

8

- 8** From the MATLAB prompt you can return control to the debugger by typing:

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type:

```
>> quit
```

- 9** When you are finished with the debugger, type:

```
<gdb> quit
```

You return to the Linux prompt.

Refer to the documentation provided with your debugger for more information on its use.

See Also

More About

- “Fortran Source MEX Files”

Handling Large mxArray

In this section...
“Using the 64-Bit API” on page 10-26
“Caution Using Negative Values” on page 10-28
“Building Cross-Platform Applications” on page 10-28

Binary MEX-files built on 64-bit platforms can handle 64-bit mxArray. These large data arrays can have up to $2^{48}-1$ elements. The maximum number of elements a sparse mxArray can have is $2^{48}-2$.

Using the following instructions creates platform-independent binary MEX-files as well.

Your system configuration can affect the performance of MATLAB. The 64-bit processor requirement enables you to create the mxArray and access data in it. However, the system memory, in particular the size of RAM and virtual memory, determine the speed at which MATLAB processes the mxArray. The more memory available, the faster the processing.

The amount of RAM also limits the amount of data you can process at one time in MATLAB. For guidance on memory issues, see “Strategies for Efficient Use of Memory”.

Using the 64-Bit API

The signatures of the API functions shown in the following table use the mwSize or mwIndex types to work with a 64-bit mxArray. The variables you use in your source code to call these functions must be the correct type.

Fortran mxArray Functions Using mwSize/mwIndex

mxCalcSingleSubscript	mxCreateStructMatrix
mxCalloc	mxGetCell
mxCopyCharacterToPtr	mxGetDimensions
mxCopyComplex16ToPtr	mxGetElementSize
mxCopyComplex8ToPtr	mxGetField
mxCopyInteger1ToPtr	mxGetFieldByNumber
mxCopyInteger2ToPtr	mxGetIr
mxCopyInteger4ToPtr	mxGetJc
mxCopyPtrToCharacter	mxGetM
mxCopyPtrToComplex16	mxGetN
mxCopyPtrToComplex8	mxGetNumberOfDimensions
mxCopyPtrToInteger1	mxGetNumberOfElements
mxCopyPtrToInteger2	mxGetNzmax
mxCopyPtrToInteger4	mxGetProperty
mxCopyPtrToPtrArray	mxGetString
mxCopyPtrToReal4	mxMalloc
mxCopyPtrToReal8	mxRealloc
mxCopyReal4ToPtr	mxSetCell
mxCopyReal8ToPtr	mxSetDimensions
mxCreateCellArray	mxSetField
mxCreateCellMatrix	mxSetFieldByNumber
mxCreateCharArray	mxSetIr
mxCreateCharMatrixFromStrings	mxSetJc
mxCreateDoubleMatrix	mxSetM
mxCreateNumericArray	mxSetN
mxCreateNumericMatrix	mxSetNzmax
mxCreateSparse	mxSetProperty
mxCreateStructArray	

Caution Using Negative Values

When using the 64-bit API, `mwSize` and `mwIndex` are equivalent to `INTEGER*8` in Fortran. This type is unsigned, unlike `INTEGER*4`, which is the type used in the 32-bit API. Be careful not to pass any negative values to functions that take `mwSize` or `mwIndex` arguments. Do not cast negative `INTEGER*4` values to `mwSize` or `mwIndex`; the returned value cannot be predicted. Instead, change your code to avoid using negative values.

Building Cross-Platform Applications

If you develop cross-platform applications (programs that can run on both 32-bit and 64-bit architectures), pay attention to the upper limit of values you use for `mwSize` and `mwIndex`. The 32-bit application reads these values and assigns them to variables declared as `INTEGER*4` in Fortran. Be careful to avoid assigning a large `mwSize` or `mwIndex` value to an `INTEGER*4` or other variable that might be too small.

Memory Management

When a MEX function returns control to MATLAB, it returns the results of its computations in the output arguments—the mxArray arrays contained in the left-side arguments `pLhs[]`. These arrays must have a temporary scope, so do not pass arrays created with the `mexMakeArrayPersistent` function in `pLhs`. MATLAB destroys any mxArray created by the MEX function that is not in `pLhs`. MATLAB also frees any memory that was allocated in the MEX function using the `mxCalloc`, `mxCalloc`, or `mxCalloc` functions.

Any misconstructed arrays left over at the end of a binary MEX file execution have the potential to cause memory errors.

MathWorks recommends that MEX 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. For more information on memory management techniques, see “Memory Management Issues” on page 7-86.

MATLAB Supports Fortran 77

MATLAB supports MEX files written in Fortran 77. When troubleshooting MEX files written in other versions of Fortran, consult outside resources.

For example, the length of the following statement is fewer than 72 characters.

```
mwPointer mxGetN, mxSetM, mxSetN, mxCreateStructMatrix, mxGetM
```

However, when MATLAB expands the preprocessor macro, `mwPointer`, the length of the statement exceeds the limit supported by Fortran 77.

Handle Complex Fortran Data

To copy complex data values between a Fortran array and a MATLAB array, call the `mxCopyComplex16ToPtr`, `mxCopyPtrToComplex16`, `mxCopyComplex8ToPtr`, and `mxCopyPtrToComplex8` functions. The example `convec.F` takes two `COMPLEX*16` row vectors and convolves them. This example uses the interleaved complex version of the Fortran Matrix API and assumes a basic understanding of MEX files as described in “Create Fortran Source MEX File” on page 10-14.

These statements copy data defined by input arrays `prhs(1)` and `prhs(2)` into Fortran variables `x` and `y` defined as `complex*16` arrays.

```
C    Load the data into Fortran arrays (native COMPLEX data).
      status =
+     mxCopyPtrToComplex16(mxGetComplexDoubles(prhs(1)),x,nx)
      if (status .ne. 1) then
          call mexErrMsgIdAndTxt (
+              'MATLAB:convec:CopyingFailed',
+              'Could not copy from prhs(1) to complex*16.')
      endif

      status =
+     mxCopyPtrToComplex16(mxGetComplexDoubles(prhs(2)),y,ny)

      if (status .ne. 1) then
          call mexErrMsgIdAndTxt (
+              'MATLAB:convec:CopyingFailed',
+              'Could not copy from prhs(2) to complex*16.')
      endif
```

Call the `convec` subroutine.

```
      call convec(x,y,z,nx,ny)
```

Copy the results into the MATLAB output array `plhs(1)`.

```
C    Load the output into a MATLAB array.
      status =
+     mxCopyComplex16ToPtr(z,mxGetComplexDoubles(plhs(1)),nz)
      if (status .ne. 1) then
          call mexErrMsgIdAndTxt (
+              'MATLAB:convec:CopyingFailed',
+              'Could not copy from complex*16 to plhs(1).')
      endif
```

Build and Test

Verify that you have an installed Fortran compiler.

```
mex -setup fortran
```

Copy the `convec.F` file to a writable folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','convec.F'),'.','f')
```

Build the file.

```
mex -R2018a convec.F
```

Test the MEX file.

```
x = [3.000 - 1.000i, 4.000 + 2.000i, 7.000 - 3.000i];  
y = [8.000 - 6.000i, 12.000 + 16.000i, 40.000 - 42.000i];  
z = convec(x,y)
```

```
z =  
    1.0e+02 *
```

```
Columns 1 through 4
```

```
0.1800 - 0.2600i 0.9600 + 0.2800i 1.3200 - 1.4400i 3.7600 - 0.1200i
```

```
Column 5
```

```
1.5400 - 4.1400i
```

Compare the results with the built-in MATLAB function `conv`.

```
conv(x,y)
```

See Also

`mxGetComplexDoubles` (Fortran)

Related Examples

- `convec.F`
- “Create Fortran Source MEX File” on page 10-14

Calling MATLAB Engine from C/C++ and Fortran Programs

- “Choosing Engine Applications” on page 11-2
- “Introducing MATLAB Engine API for C and Fortran” on page 11-4
- “Callbacks in Applications” on page 11-6
- “Call MATLAB Functions from C Applications” on page 11-7
- “Call MATLAB Functions from Fortran Applications” on page 11-9
- “Attach to Existing MATLAB Sessions” on page 11-11
- “Build Windows Engine Application” on page 11-13
- “Run Windows Engine Application” on page 11-14
- “Set Run-Time Library Path on Windows Systems” on page 11-15
- “Register MATLAB as a COM Server” on page 11-17
- “Build Linux Engine Application” on page 11-18
- “Build macOS Engine Application” on page 11-19
- “Run Linux Engine Application” on page 11-20
- “Run macOS Engine Application” on page 11-21
- “Set Run-Time Library Path on Linux Systems” on page 11-22
- “Set Run-Time Library Path on macOS Systems” on page 11-23
- “Build Engine Applications with IDE” on page 11-25
- “Can't Start MATLAB Engine” on page 11-28
- “Multithreaded Applications” on page 11-29
- “User Input Not Supported” on page 11-30
- “Getting Started” on page 11-31
- “Debug MATLAB Function Called by C Engine” on page 11-32

Choosing Engine Applications

You can call MATLAB functions from your own external language programs. These programs are called engine applications. To create an engine application, write your program using MATLAB APIs then build using the mex command.

MATLAB supports engine applications written in C, C++, Java, Fortran, or Python.

To use C++ 11 programming features, see:

- “MATLAB Engine API for C++”
- “MATLAB Data API”

If your C engine applications must run in MATLAB R2017b or earlier, or if you prefer to work in the C language, then use functions based on the C Matrix API.

- “MATLAB Engine API for C”
- “C Matrix API”
- C MEX API (optional)

Caution Do not mix functions in different C APIs. For example, do not use functions in the C Matrix API with functions in the MATLAB Data API.

To write Java engine applications, see “Calling MATLAB from Java”.

To write Fortran engine applications, see:

- “MATLAB Engine API for Fortran”
- “Fortran Matrix API”
- “Fortran MEX API” (optional)

To write Python engine applications, see “Calling MATLAB from Python”.

See Also

More About

- “MATLAB Engine API for C++”
- “MATLAB Engine API for C”
- “Calling MATLAB from Java”
- “MATLAB Engine API for Fortran”
- “Calling MATLAB from Python”

Introducing MATLAB Engine API for C and Fortran

The MATLAB C and Fortran engine library contains routines that allow you to call MATLAB from your own programs, using MATLAB as a computation engine. Using the MATLAB engine requires an installed version of MATLAB; you cannot run the MATLAB engine on a machine that only has the MATLAB Runtime.

Engine programs are standalone programs. These programs communicate with a separate MATLAB process via pipes, on UNIX systems, and through a Microsoft Component Object Model (COM) interface, on Microsoft Windows systems. MATLAB provides a library of functions that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB.

Some of the things you can do with the MATLAB engine are:

- Call a math routine, for example, to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.
- Build an entire system for a specific task. For example, the front end (user interface) is programmed in C and the back end (analysis) is programmed in MATLAB.

The MATLAB engine operates by running in the background as a separate process from your own program. Some advantages are:

- On UNIX systems, the engine can run on your machine, or on any other UNIX machine on your network, including machines of a different architecture. This configuration allows you to implement a user interface on your workstation and perform the computations on a faster machine located elsewhere on your network. For more information, see the `engOpen` reference page.
- Instead of requiring your program to link to the entire MATLAB program (a substantial amount of code), it links to a smaller engine library.

The MATLAB engine cannot read MAT-files in a format based on HDF5. These MAT-files save data using the `-v7.3` option of the `save` function or are opened using the `w7.3` mode argument to the C or Fortran `matOpen` function.

Note To run MATLAB engine on the UNIX platform, you must have the C shell `csh` installed at `/bin/csh`.

Communicating with MATLAB Software

On UNIX systems, the engine library communicates with the engine using pipes, and, if needed, `rsh` for remote execution. On Microsoft Windows systems, the engine library communicates with the engine using a Component Object Model (COM) interface.

See Also

Related Examples

- “Call MATLAB Functions from C Applications” on page 11-7

Callbacks in Applications

If you have a user interface that executes many callbacks through the MATLAB engine, force these callbacks to be evaluated in the context of the base workspace. Use `evalin` to specify the base workspace for evaluating the callback expression. For example:

```
engEvalString(ep,"evalin('base', expression)")
```

Specifying the base workspace ensures MATLAB processes the callback correctly and returns results for that call.

This advice does not apply to computational applications that do not execute callbacks.

See Also

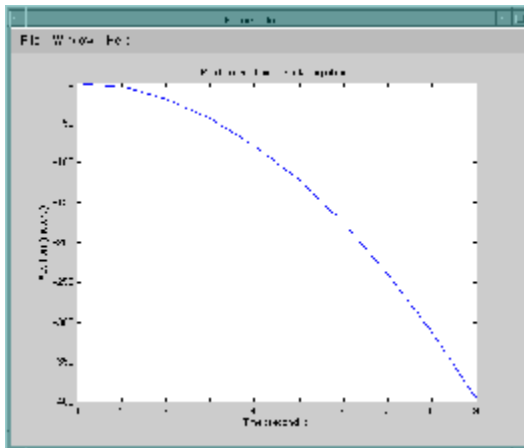
`engEvalString` | `evalin`

Call MATLAB Functions from C Applications

The program `engdemo.c` in the `matlabroot/extern/examples/eng_mat` folder illustrates how to call the engine functions from a standalone C program.

For a Microsoft Windows version of this program, open `engwindemo.c` in the `matlabroot\extern\examples\eng_mat` folder. For a C++ version, open `engdemo.cpp`.

The first part of this program starts MATLAB and sends it data. MATLAB analyzes the data and plots the results.



The program continues with:

Press Return to continue

Pressing **Return** continues the program:

Done for Part I.

Enter a MATLAB command to evaluate. This command should create a variable `X`. This program will then determine what kind of variable you created.

For example: `X = 1:5`

Entering `X = 17.5` continues the program execution.

`X = 17.5`

```
X =  
    17.5000
```

```
Retrieving X...  
X is class double  
Done!
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

See Also

Related Examples

- “Build Windows Engine Application” on page 11-13
- “Build Linux Engine Application” on page 11-18

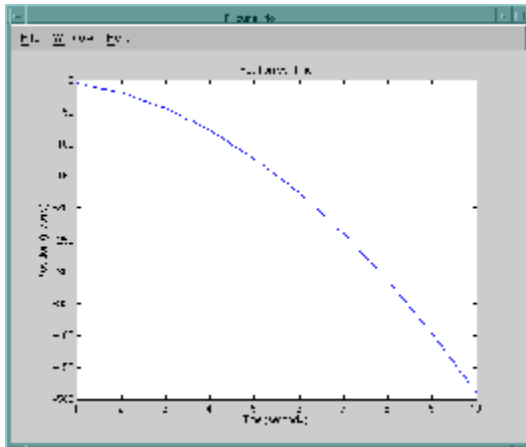
More About

- “Build Engine Applications with IDE” on page 11-25

Call MATLAB Functions from Fortran Applications

The program `fengdemo.F`, in the `matlabroot/extern/examples/eng_mat` folder, illustrates how to call the engine functions from a standalone Fortran program. To see the code, open this file.

Executing this program starts MATLAB, sends it data, and plots the results.



The program continues with:

```
Type 0 <return> to Exit
Type 1 <return> to continue
```

Entering 1 at the prompt continues the program execution:

```
1
MATLAB computed the following distances:
time(s)  distance(m)
1.00     -4.90
2.00     -19.6
3.00     -44.1
4.00     -78.4
5.00     -123.
6.00     -176.
7.00     -240.
8.00     -314.
9.00     -397.
10.0     -490.
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Attach to Existing MATLAB Sessions

This example shows how to attach an engine program to a MATLAB session that is already running.

On a Windows platform, start MATLAB with `-automation` in the command line. When you call `engOpen`, it connects to this existing session. Call `engOpen` only once, because any `engOpen` calls now connect to this one MATLAB session.

The `-automation` option also causes the command window to be minimized. You must open it manually.

On the macOS and Linux platforms, you cannot make an engine program connect to an existing MATLAB session.

Shut down any MATLAB sessions.

From the **Start** button on the Windows menu bar, click **Run**.

In MATLAB, capture the value returned by the command:

```
path = fullfile(matlabroot, 'bin', computer('arch'))
```

In the **Run** dialog box **Open** field, type the following command, replacing *path* with the value from the previous step:

```
path\matlab.exe -automation
```

To start MATLAB, click **OK**.

In MATLAB, copy the `engwindemo.c` example to a writable folder.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'engwindemo.c'), '.', 'f')
```

Build the example.

```
mex -client engine engwindemo.c
```

Run the `engwindemo` program by typing at the MATLAB prompt:

```
!engwindemo
```

This command does not start another MATLAB session, but rather uses the MATLAB session that is already open.

See Also

More About

- “Start MATLAB as Automation Server in Desktop Mode” on page 18-14
- “MATLAB COM Integration” on page 16-2

Build Windows Engine Application

This example shows how to verify the build process on Windows platforms using the C example `engwindemo.c`.

Start MATLAB as a user with administrator privileges. Based on your User Account Control (UAC) settings, you might need to right-click the MATLAB icon and select **Run as administrator**. If that option is not available, contact your system administrator.

Register MATLAB as a COM server.

```
!matlab -regserver
```

MATLAB displays a second, minimized command window. Close that window.

Verify your current folder is writable and copy the example.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'engwindemo.c'), '.', 'f')
```

Choose a compiler, if necessary.

```
mex -setup -client engine C
```

Build the application.

```
mex -v -client engine engwindemo.c
```

See Also

`mex`

Related Examples

- “Call MATLAB Functions from C Applications” on page 11-7
- “Run Windows Engine Application” on page 11-14

More About

- “Register MATLAB as a COM Server” on page 11-17

Run Windows Engine Application

This example shows how to run the C example, `engwindemo.c`, from the Windows system prompt. Make note of the value of `matlabroot` (the folder where MATLAB is installed) and the value returned by the MATLAB `computer('arch')` command (`arch`).

Set the run-time library path by modifying the system PATH variable.

```
set PATH=matlabroot\bin\arch;%PATH%
```

Make sure that you include the `;` path terminator character.

Run the example. The `engwindemo` application must be on your system path.

```
engwindemo
```

MATLAB starts and displays the results.

To close the application, click **Ok** in the MATLAB `whos` window.

See Also

`computer` | `matlabroot`

Related Examples

- “Build Windows Engine Application” on page 11-13
- “Call MATLAB Functions from C Applications” on page 11-7

More About

- “Set Run-Time Library Path on Windows Systems” on page 11-15

Set Run-Time Library Path on Windows Systems

In this section...

“Change Path Each Time You Run the Application” on page 11-15

“Permanently Change Path” on page 11-15

“Troubleshooting” on page 11-16

At run time, tell the operating system where the API shared libraries reside by setting the Path environment variable. Set the value to the path returned by the following MATLAB command:

```
res = fullfile(matlabroot, 'bin', computer('arch'))
```

Change Path Each Time You Run the Application

To set the run-time library path from the Windows command prompt, type the following command, where *res* is the value returned from the `fullfile` command. Set the path every time that you open the Windows Command Processor.

```
set PATH=res; %PATH%
```

Set the path each time that you open the Windows prompt.

Permanently Change Path

To set the run-time library path permanently to *res*, modify the Path variable using the Control Panel. For the setting to take effect, close the command prompt window, then open a new command prompt window.

To remove the run-time library path, follow the same instructions, deleting the path name from the variable.

Windows 7

- Select **Computer** from the Start menu.
- Choose **System properties** from the context menu.
- Click **Advanced system settings > Advanced** tab.
- Click **Environment Variables**.

- Under **System variables**, select Path and click **Edit**.
- Modify Path by inserting *res;* at the beginning of the **Variable value:** text field.
- To close the dialog boxes, click **Ok** then close the **Control Panel** dialog box.

Troubleshooting

If you have multiple versions of MATLAB installed on your system, the version you use to build your engine applications must be the first listed in your system Path environment variable. Otherwise, MATLAB displays Can't start MATLAB engine.

Register MATLAB as a COM Server

To run the engine application on a Windows operating system, you must register MATLAB as a COM server. Register MATLAB for every session, to ensure that the current version of MATLAB is the registered version. If you run older versions, the registered version could change. If there is a mismatch of version numbers, MATLAB displays Can't start MATLAB engine.

To register MATLAB manually as a server, start MATLAB as a user with administrator privilege. Then type:

```
!matlab -regserver
```

Close the MATLAB window that appears.

Build Linux Engine Application

This example shows how to verify the build process on a Linux platform using the C example `engdemo.c`.

Open MATLAB.

Verify your current folder is writable and copy the example.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'engdemo.c'), '.', 'f')
```

Build the application.

```
mex -v -client engine engdemo.c
```

See Also

`mex`

Related Examples

- “Call MATLAB Functions from C Applications” on page 11-7
- “Run Linux Engine Application” on page 11-20

Build macOS Engine Application

This example shows how to verify the build process on a macOS platform using the C example `engdemo.c`.

Open MATLAB.

Verify your current folder is writable and copy the example.

```
copyfile(fullfile(matlabroot, 'extern', 'examples', 'eng_mat', 'engdemo.c'), '.', 'f')
```

Build the application.

```
mex -v -client engine engdemo.c
```

See Also

`mex`

Related Examples

- “Call MATLAB Functions from C Applications” on page 11-7
- “Run macOS Engine Application” on page 11-21

Run Linux Engine Application

This example shows how to run the C example `engdemo.c` from the Linux system prompt. Make note of the value of `matlabroot`, the folder where MATLAB is installed.

Set the run-time library path. This command replaces the value, if any, in `LD_LIBRARY_PATH`.

```
setenv LD_LIBRARY_PATH matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64
```

Set the path. Make sure that you include the `:` path terminator character.

```
setenv PATH matlabroot/bin:$PATH
```

Run the example. The `engdemo` application must be on your system path.

```
./engdemo
```

MATLAB starts and displays a figure.

To close the figure, press Return.

Create variable `X`, for example:

```
X = 'hello'
```

MATLAB displays the results and closes.

See Also

Related Examples

- “Build Linux Engine Application” on page 11-18
- “Call MATLAB Functions from C Applications” on page 11-7

More About

- “Set Run-Time Library Path on Linux Systems” on page 11-22

Run macOS Engine Application

This example shows how to run the C example `engdemo.c` from the macOS Terminal Window. Make note of the value of `matlabroot`, the folder where MATLAB is installed.

Set the run-time library path. This command replaces the value, if any, in `DYLD_LIBRARY_PATH`. For more information, see “Set Run-Time Library Path on macOS Systems” on page 11-23.

```
setenv DYLD_LIBRARY_PATH matlabroot/bin/maci64:matlabroot/sys/os/maci64
```

Set the path. Make sure that you include the `:` path terminator character.

```
setenv PATH matlabroot/bin:$PATH
```

Run the example. The `engdemo` application must be on your system path.

```
./engdemo
```

MATLAB starts and displays a figure.

To close the figure, press Return.

Create variable `X`, for example:

```
X = 'hello'
```

MATLAB displays the results and closes.

See Also

Related Examples

- “Build macOS Engine Application” on page 11-19
- “Call MATLAB Functions from C Applications” on page 11-7

More About

- “Set Run-Time Library Path on macOS Systems” on page 11-23

Set Run-Time Library Path on Linux Systems

At run time, tell the operating system where the API shared libraries reside by setting the environment variable `LD_LIBRARY_PATH`. Set the value to `matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64`.

The command you use depends on your shell. The command replaces the existing `LD_LIBRARY_PATH` value. If `LD_LIBRARY_PATH` is already defined, prepend the new value to the existing value.

If you have multiple versions of MATLAB installed on your system, the version you use to build your engine applications must be the first listed in your system `Path` environment variable. Otherwise, MATLAB displays `Can't start MATLAB engine`.

Set the path every time you run MATLAB. Alternatively, place the commands in a MATLAB startup script.

C Shell

Set the library path using the following command format.

```
setenv LD_LIBRARY_PATH matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64
```

You can place these commands in a startup script, such as `~/ .cshrc`.

Bourne Shell

Set the library path using the following command format.

```
LD_LIBRARY_PATH=matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64:LD_LIBRARY_PATH  
export LD_LIBRARY_PATH
```

You can place these commands in a startup script such as `~/ .profile`.

See Also

`matlabroot`

Set Run-Time Library Path on macOS Systems

At run time, tell the operating system where the API shared libraries reside by setting the environment variable `DYLD_LIBRARY_PATH`. Set the value to `matlabroot/bin/maci64:matlabroot/sys/os/maci64`.

The command you use depends on your shell. The command replaces the existing `DYLD_LIBRARY_PATH` value. If `DYLD_LIBRARY_PATH` is already defined, prepend the new value to the existing value.

If you have multiple versions of MATLAB installed on your system, the version you use to build your engine applications must be the first listed in your system `Path` environment variable. Otherwise, MATLAB displays `Can't start MATLAB engine`.

Set the path every time you run MATLAB. Alternatively, place the commands in a MATLAB startup script.

C Shell

Set the library path using the following command format.

```
setenv DYLD_LIBRARY_PATH matlabroot/bin/maci64:matlabroot/sys/os/maci64
```

For example, for MATLAB R2015a on a Mac system:

```
setenv DYLD_LIBRARY_PATH /Applications/MATLAB_R2015a.app/bin/maci64:/Applications/MATLAB_R2015a.app/sys/os/maci64
```

You can place these commands in a startup script, such as `~/ .cshrc`.

Bourne Shell

Set the library path using the following command format.

```
DYLD_LIBRARY_PATH=matlabroot/bin/maci64:matlabroot/sys/os/maci64:DYLD_LIBRARY_PATH  
export DYLD_LIBRARY_PATH
```

For example, for MATLAB R2015a on a Mac system:

```
DYLD_LIBRARY_PATH=/Applications/MATLAB_R2015a.app/bin/maci64:/Applications/MATLAB_R2015a.app/sys/os/maci64:$DYLD_LIBRARY_PATH  
export DYLD_LIBRARY_PATH
```

You can place these commands in a startup script such as `~/ .profile`.

See Also

matlabroot

External Websites

- Append library path to "DYLD_LIBRARY_PATH" in MAC

Build Engine Applications with IDE

In this section...

“Configuring the IDE” on page 11-25

“Engine Include Files” on page 11-25

“Engine Libraries” on page 11-26

“Library Files Required by libeng” on page 11-27

You can use the MATLAB Editor to write your engine application code and the `mex` command to build it. If you prefer to use an integrated development environment (IDE) such as Microsoft Visual Studio or Xcode to write your source code, you can still use the `mex` command. However, to build your application with your IDE, follow the guidelines in the following topics.

Configuring the IDE

To use your integrated development environment to build engine applications, your IDE needs a MATLAB-supported compiler. For an up-to-date list of supported compilers, see [Supported and Compatible Compilers](#).

Engine applications require the Engine Library `libeng`, the Matrix Library `libmx`, and supporting include files. When you build using the `mex` command, MATLAB is configured to locate these files. When you build in your IDE, you must configure the IDE to locate them. Where these settings are depends on your IDE. Refer to your product documentation.

Engine Include Files

Header files contain function declarations with prototypes for the routines you access in the API libraries. These files are in the `matlabroot\extern\include` folder and are the same for Windows, macOS, and Linux systems. Engine applications use:

- `engine.h` — Function prototypes for engine routines
- `matrix.h` — Definition of the `mxArray` structure and function prototypes for matrix access routines
- `mat.h` (optional) — Function prototypes for `mat` routines

In your IDE, set the pre-processor include path to the value returned by the following MATLAB command:

```
fullfile(matlabroot, 'extern', 'include')
```

Engine Libraries

You need the `libeng` and `libmx` shared libraries. The name of the file is platform-specific. Add these library names to your IDE configuration. Refer to your IDE product documentation for instructions.

Windows Libraries

In these path specifications, replace *compiler* with either `microsoft` or `mingw64`.

- Engine library — `matlabroot\extern\lib\win64\compiler\libeng.lib`
- Matrix library — `matlabroot\extern\lib\win64\compiler\libmx.lib`
- MEX library (optional) — `matlabroot\extern\lib\win64\compiler\libmex.lib`
- MAT-File library (optional) — `matlabroot\extern\lib\win64\compiler\libmat.lib`

Linux Libraries

- Engine library — `matlabroot/bin/glnxa64/libeng.so`
- Matrix library — `matlabroot/bin/glnxa64/libmx.so`
- MEX library (optional) — `matlabroot/bin/glnxa64/libmex.so`
- MAT-File library (optional) — `matlabroot/bin/glnxa64/libmat.so`

macOS Libraries

- Engine library — `matlabroot/bin/maci64/libeng.dylib`
- Matrix library — `matlabroot/bin/maci64/libmx.dylib`
- MEX library (optional) — `matlabroot/bin/maci64/libmex.dylib`
- MAT-File library (optional) — `matlabroot/bin/maci64/libmat.dylib`

Library Files Required by libeng

The `libeng` library requires additional third-party library files. MATLAB uses these libraries to support Unicode character encoding and data compression in MAT-files.

These library files must reside in the same folder as the `libmx` library. You can determine what these libraries are using the platform-specific commands shown in the following table. Once you identify these files, update your IDE.

Library Dependency Commands

Windows	Linux	macOS
See the following instructions for Dependency Walker	<code>ldd -d libeng.so</code>	<code>otool -L libeng.dylib</code>

To find library dependencies on Windows systems, use the third-party product Dependency Walker. This free utility scans Windows modules and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions exported by that module, and which of those functions are called by other modules. See [How do I determine which libraries my MEX-file or stand-alone application requires?](#) for information on using the Dependency Walker.

Drag and drop the `libeng.dll` file into the Depends window. Identify the dependent libraries and add them to your IDE configuration.

See Also

Related Examples

- “Build Windows Engine Application” on page 11-13
- “Build Linux Engine Application” on page 11-18

External Websites

- [How can I compile a MATLAB Engine application using Microsoft Visual Studio 9.0 or 10.0?](#)
- [How can I build an Engine application using the Xcode IDE on Mac?](#)

Can't Start MATLAB Engine

Set the run-time library path to tell the operating system the location of the API shared libraries.

If you have multiple versions of MATLAB installed on your system, the version you use to build your engine applications must be the first listed in your system Path environment variable. Otherwise, MATLAB displays `Can't start MATLAB engine`.

On Windows operating systems, you also must register MATLAB as a COM server. If you have multiple versions of MATLAB, the version you are using must be the registered version.

On a UNIX platform, you must have the C shell `cs` installed at `/bin/csh`.

See Also

Related Examples

- “Register MATLAB as a COM Server” on page 11-17
- “Set Run-Time Library Path on Windows Systems” on page 11-15
- “Set Run-Time Library Path on macOS Systems” on page 11-23
- “Set Run-Time Library Path on Linux Systems” on page 11-22

Multithreaded Applications

MATLAB libraries are not thread-safe. If you create multithreaded applications, make sure that only one thread accesses the engine application.

User Input Not Supported

A MATLAB engine application runs MATLAB as a computational server. MATLAB functions that interact with the user are not supported. Examples of such functions include input, keyboard, and pause.

For debugging information, see “Debug MATLAB Function Called by C Engine” on page 11-32.

See Also

More About

- “Debug MATLAB Function Called by C Engine” on page 11-32

Getting Started

To build a C engine application, you need:

- The ability to write C or Fortran source code. You can create these files with the MATLAB Editor.
- A compiler supported by MATLAB. For an up-to-date list of supported compilers, see Supported and Compatible Compilers.
- “C Matrix API” or “Fortran Matrix API” functions. However, the `mxGetProperty` and `mxSetProperty` functions are not supported for standalone applications.
- C and Fortran Engine Library functions.
- The `mex` build script with the `-client engine` option.
- To use your own build tools, see “Build Engine Applications with IDE” on page 11-25.

To run the application:

- “Set Run-Time Library Path on Windows Systems” on page 11-15
- “Register MATLAB as a COM Server” on page 11-17
- “Set Run-Time Library Path on macOS Systems” on page 11-23
- “Set Run-Time Library Path on Linux Systems” on page 11-22

See Also

“MATLAB Engine API for C++”

Debug MATLAB Function Called by C Engine

When creating MATLAB functions for use in engine applications, it is good practice to run the functions in MATLAB before calling them through the engine library functions.

After you integrate a MATLAB function into an engine application, you can use the `dbstop` and other MATLAB debugging commands to debug the function invoked by the engine application. You might need to add code to the application to pause processing so that you can enter a debug command, unless you attach the application to an existing MATLAB session.

Debug Engine Application on Windows

Assume that you have an engine application that calls a MATLAB function `myfcn` on Windows. Add the following statement to your engine code before the call to `myfcn`. This code waits for user input.

```
fgetc(stdin);
```

Start the engine application. The application opens a MATLAB desktop and a Windows console application of MATLAB. At the desktop command prompt, type the following command. Make sure that the function is on your MATLAB path.

```
dbstop("myfcn")
```

In the console window, press **Enter**. MATLAB enters debug mode; type debug commands in the desktop command window.

Debug Engine Application Attached to Existing MATLAB Session

For information about using an existing MATLAB session, see “Attach to Existing MATLAB Sessions” on page 11-11. To debug function `myfcn` called by an engine application, first start MATLAB as an Automation server with one of the following commands.

- From the system prompt:

```
matlab.exe -automation
```

- From the MATLAB command prompt:

```
state = enableservice('AutomationServer',true);
```

At the MATLAB command prompt, type:

```
dbstop("myfcn")
```

Start the engine application. MATLAB enters debug mode; type debug commands in the MATLAB command window.

See Also

Engine API for Java

- “MATLAB Engine API for Java” on page 12-2
- “Build Java Engine Programs” on page 12-3
- “Java Example Source Code” on page 12-7
- “Java Engine API Summary” on page 12-8
- “Java Data Type Conversions” on page 12-11
- “Start and Close MATLAB Session from Java” on page 12-14
- “Connect Java to Running MATLAB Session” on page 12-17
- “Execute MATLAB Functions from Java” on page 12-20
- “Evaluate MATLAB Statements from Java” on page 12-23
- “Pass Variables from Java to MATLAB” on page 12-25
- “Pass Variables from MATLAB to Java” on page 12-27
- “Using Complex Variables in Java” on page 12-28
- “Using MATLAB Structures in Java” on page 12-31
- “Pass Java CellStr to MATLAB” on page 12-34
- “Using MATLAB Handle Objects in Java” on page 12-36
- “Redirect MATLAB Command Window Output to Java” on page 12-38
- “Run Simulink Simulation from Java” on page 12-40
- “MATLAB Engine API Exceptions” on page 12-44
- “Pass Java Array Arguments to MATLAB” on page 12-46
- “Incorrect Java Data Types” on page 12-48
- “Incorrect Number of Output Arguments” on page 12-50

MATLAB Engine API for Java

The MATLAB Engine API for Java enables Java programs to interact with MATLAB synchronously or asynchronously, including:

- Start and terminate MATLAB.
- Connect to and disconnect from MATLAB sessions on the local machine.
- Call MATLAB functions with input arguments passed from Java and output variables returned from MATLAB.
- Evaluate MATLAB statements in the MATLAB base workspace.
- Pass variables from Java to MATLAB and MATLAB to Java.

Asynchronous communication with MATLAB is based on the Java Future interface, `java.util.concurrent.Future`.

The size of data arrays passed between Java and MATLAB is limited to 2 GB. This limit applies to the data plus supporting information passed between the processes.

The MATLAB Engine API for Java is included as part of the MATLAB product. You must have JDK 7 or 8 installed to build a MATLAB Engine application for Java.

See Also

Related Examples

- “Java Engine API Summary” on page 12-8
- “Build Java Engine Programs” on page 12-3
- “Java Example Source Code” on page 12-7
- “Start and Close MATLAB Session from Java” on page 12-14

Build Java Engine Programs

In this section...

- “General Requirements” on page 12-3
- “Compile and Run Java Code on Windows” on page 12-4
- “Compile and Run Java Code on macOS” on page 12-4
- “Compile and Run Java Code on Linux” on page 12-5

General Requirements

To set up your Java environment for building engine applications:

- Add `matlabroot/extern/engines/java/jar/engine.jar` to your Java class path.
- Build the engine application with JDK 7 or 8.
- Ensure your JRE™ is not an earlier version than your JDK.

To run Java, add the folder `matlabroot/bin/<arch>` to your system environment variable. `<arch>` is your computer architecture. For example, `win64` for 64-bit Microsoft Windows machines, `maci64` on macOS, or `glnxa64` on Linux.

`matlabroot` is the value returned by the MATLAB `matlabroot` command. This command returns the folder where MATLAB is installed.

This table lists the names of the environment variables and the values of the paths.

Operating System	Variable	Path
Windows	PATH	<code>matlabroot\bin\win64</code>
64-bit Apple Mac	DYLD_LIBRARY_PATH	<code>matlabroot/bin/maci64</code>
64-bit Linux	LD_LIBRARY_PATH	<code>matlabroot/bin/ glnxa64:matlabroot/sys/os/ glnxa64</code>

Compile and Run Java Code on Windows

Compile your Java code:

```
javac -classpath matlabroot\extern\engines\java\jar\engine.jar MyJavaCode.java
```

Run the Java program:

```
java -classpath .;matlabroot\extern\engines\java\jar\engine.jar MyJavaCode
```

Set System Path

To set the runtime library path from the Windows command prompt, type the following command.

```
set PATH=matlabroot\bin\win64;%PATH%
```

Set the path every time you open the Windows Command Processor.

You can also set the PATH variable from the System Properties dialog box. From the **Control Panel > System > Advanced system settings > Advanced** tab, click **Environment Variables**. Under **System variables**, select Path and click **Edit**. Modify Path by inserting *matlabroot\bin\win64*; at the beginning of the **Variable Value**. Click **OK** to close the dialog boxes, then close the **Control Panel** dialog box.

Compile and Run Java Code on macOS

MATLAB engine API for Java supports only maci64 on macOS systems.

Compile the Java code:

```
javac -classpath matlabroot/extern/engines/java/jar/engine.jar MyJavaCode.java
```

Specify Java Library Path and Run Program

Specify the Java library path and run Java program in one statement.

```
java -Djava.library.path=matlabroot/bin/maci64 -classpath .:matlabroot/extern/engines/java/jar/engine.jar MyJavaCode
```

Set System Variable and Run Program

Set the DYLD_LIBRARY_PATH variable and run Java program. For example, using a C shell:

```
setenv DYLD_LIBRARY_PATH matlabroot/bin/maci64:$DYLD_LIBRARY_PATH  
java -classpath .:matlabroot/extern/engines/java/jar/engine.jar MyJavaCode
```

Set Variables from C Shell

You can put these commands in a startup script, such as `~/ .cshrc`.

```
setenv DYLD_LIBRARY_PATH matlabroot/bin/maci64:$DYLD_LIBRARY_PATH
```

Set Variables in Bourne Shell

You can put these commands in a startup script such as `~/ .profile`.

```
DYLD_LIBRARY_PATH=matlabroot/bin/maci64:$DYLD_LIBRARY_PATH  
export DYLD_LIBRARY_PATH
```

Using Early Builds of Java Version 1.8.0

When using early builds of Java version 1.8.0, such as 1.8.0_111, the `DYLD_LIBRARY_PATH` environment variable might not be recognized. If you receive a `java.lang.UnsatisfiedLinkError` exception, set the `java.library.path` explicitly:

```
java -Djava.library.path=matlabroot/bin/maci64 -classpath .:matlabroot/extern/engines/java/jar/engine.jar MyJavaCode
```

Compile and Run Java Code on Linux

MATLAB engine API for Java supports only `glnxa64` on Linux systems.

Compile Java code:

```
javac -classpath matlabroot/extern/engines/java/jar/engine.jar MyJavaCode.java
```

Specify Java Library Path and Run Program

If a compatible GCC library is in the search path, you can add `matlabroot/bin/glnxa64` to the Java library search path and run the examples without setting the `LD_LIBRARY_PATH` variable. For information on supported compilers, see Supported and Compatible Compilers.

Specify the Java library path and run the Java program in one statement.

```
java -Djava.library.path=matlabroot/bin/glnxa64 -classpath .:matlabroot/extern/engines/java/jar/engine.jar MyJavaCode
```

Set System Variable and Run Program

Set the `LD_LIBRARY_PATH` variable and run Java program. For example, using a C shell:

```
setenv LD_LIBRARY_PATH matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64:$LD_LIBRARY_PATH
java -classpath .:matlabroot/extern/engines/java/jar/engine.jar MyJavaCode
```

Set Variables from C Shell

You can put these commands in a startup script, such as `~/ .cshrc`.

```
setenv LD_LIBRARY_PATH matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64:$LD_LIBRARY_PATH
```

Set Variables from Bourne Shell

You can put these commands in a startup script such as `~/ .profile`.

```
LD_LIBRARY_PATH=matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

See Also

Related Examples

- “Java Engine API Summary” on page 12-8

Java Example Source Code

The MATLAB engine API for Java ships with example code that you can use to test your environment and to become familiar with the use of the engine API. The Java source code is located in the `matlabroot/extern/examples/engines/java` folder, where `matlabroot` is the path returned by the MATLAB `matlabroot` command.

The folder contains these files:

- `EngineGUIDemo.java` — A Swing-based user interface that accepts an input number and uses MATLAB to calculate the factorial of the number.
- `EngineConsoleDemo.java` — A Java program that uses MATLAB logical indexing and mathematical functions to operate on a numeric matrix.
- `README` — A text file that describes how to build and run the examples.

To build these examples, first copy the files to a writable folder on your path. You can use the MATLAB `copyfile` function for this purpose:

```
copyfile(fullfile(matlabroot,'extern','examples','engines','java','EngineGUIDemo.java'),'DestinationFolder')
copyfile(fullfile(matlabroot,'extern','examples','engines','java','EngineConsoleDemo.java'),'DestinationFolder')
```

Follow the instructions in the `README` file to compile and run the examples.

See Also

Related Examples

- “Build Java Engine Programs” on page 12-3

Java Engine API Summary

In this section...
“com.mathworks Packages” on page 12-8
“com.mathworks.engine.MatlabEngine Methods” on page 12-9
“java.util.concurrent.Future Interface” on page 12-10

com.mathworks Packages

Classes in com.mathworks Package	Purpose
com.mathworks.engine.MatlabEngine	Definition of the API for the Java engine
com.mathworks.engine.EngineException on page 12-44	Failure by MATLAB to start, connect, terminate, or disconnect
com.mathworks.engine.UnsupportedTypeException on page 12-44	Unsupported data type in input or output of MATLAB function
com.mathworks.engine.MatlabExecutionException on page 12-44	Runtime error in MATLAB code
com.mathworks.engine.MatlabSyntaxException on page 12-44	Syntax error in MATLAB expression

The com.mathworks.matlab.types package provides support for specialized MATLAB types in Java.

Classes in com.mathworks.matlab.types Package	MATLAB Type
com.mathworks.matlab.types.Complex	MATLAB complex values in Java
com.mathworks.matlab.types.HandleObject	MATLAB handle objects in Java
com.mathworks.matlab.types.Struct	MATLAB struct (structures) in Java
com.mathworks.matlab.types.CellStr	Create a cell array of characters to pass to MATLAB

com.mathworks.engine.MatlabEngine Methods

Static methods	Purpose
"startMatlab"	Start MATLAB synchronously
"startMatlabAsync"	Start MATLAB asynchronously
"findMatlab"	Find all available shared MATLAB sessions running on the local machine synchronously
"findMatlabAsync"	Find all available shared MATLAB sessions from local machine asynchronously
"connectMatlab"	Connect to a shared MATLAB session on local machine synchronously
"connectMatlabAsync"	Connect to a shared MATLAB session on local machine asynchronously

Member Methods	Purpose
"feval"	Evaluate a MATLAB function with arguments synchronously
"fevalAsync"	Evaluate a MATLAB function with arguments asynchronously
"eval"	Evaluate a MATLAB statement as a string synchronously
"evalAsync"	Evaluate a MATLAB statement as a string asynchronously
"getVariable"	Get a variable from the MATLAB base workspace synchronously
"getVariableAsync"	Get a variable from the MATLAB base workspace asynchronously
"putVariable"	Put a variable in the MATLAB base workspace synchronously
"putVariableAsync"	Put a variable in the MATLAB base workspace asynchronously
"disconnect"	Explicitly disconnect from the current MATLAB session synchronously

Member Methods	Purpose
"disconnectAsync"	Explicitly disconnect from the current MATLAB session asynchronously
"quit"	Force the shutdown of the current MATLAB session synchronously
"quitAsync"	Force the shutdown of the current MATLAB session asynchronously
"close"	Disconnect or terminate current MATLAB session

java.util.concurrent.Future Interface

Member Methods	Purpose
get	Wait for the computation to complete and then return the result
cancel	Attempt to cancel execution of this task
isCancelled	Return <code>true</code> if this task was cancelled before it completed
isDone	Return <code>true</code> if this task completes

For more information, see the Java documentation for `java.util.concurrent.Future`.

Java Data Type Conversions

In this section...

“Pass Java Data to MATLAB” on page 12-11

“Pass MATLAB Data to Java” on page 12-12

Pass Java Data to MATLAB

This table shows how the MATLAB engine API maps Java data types to MATLAB data types.

Java Type	MATLAB Type for Scalar Data	MATLAB Type for Array Data
boolean	logical	logical
bytes	int8	int8
short	int16	int16
int	int32	int32
long	int64	int64
float	single	single
double	double	double
char	char	char
java.lang.String	string	string
com.mathworks.matlab.types.Struct	struct	struct
com.mathworks.matlab.types.Complex	double	double
com.mathworks.matlab.types.HandleObject	MATLAB handle object	MATLAB handle object
com.mathworks.matlab.types.CellStr	cellstr	cellstr
Nonrectangular (jagged) array	N/A	cell

Pass MATLAB Data to Java

This table shows how the MATLAB engine API maps MATLAB data types to Java data types.

MATLAB Type	Java Type for Scalar Data	Java Type for Array Data
logical	Boolean	boolean[]
int8	Byte	byte[]
uint8	Byte	byte[]
int16	Short	short[]
uint16	Short	short[]
int32	Integer	int[]
uint32	Integer	int[]
int64	Long	long[]
uint64	Long	long[]
single	Float	float[]
double	Double	double[]
char	String	String
string	String	String[]
cellstr	String	String[]
Mixed cell array	N/A	Object[]
MATLAB handle object	com.mathworks.matlab.types.HandleObject	com.mathworks.matlab.types.HandleObject[]
MATLAB value object	Unsupported	Unsupported
Complex number	com.mathworks.matlab.types.Complex	com.mathworks.matlab.types.Complex[]
struct	com.mathworks.matlab.types.Struct	com.mathworks.matlab.types.Struct[]

See Also

More About

- “Java Engine API Summary” on page 12-8

Start and Close MATLAB Session from Java

In this section...
“Start MATLAB Synchronously” on page 12-14
“Start MATLAB Asynchronously” on page 12-14
“Start Engine with Startup Options” on page 12-15
“Close MATLAB Engine Session” on page 12-15

You can start a MATLAB session from your Java program synchronously or asynchronously. Use these `MatlabEngine` static methods to start MATLAB:

- `MatlabEngine.startMatlab` — Start a MATLAB session synchronously.
- `MatlabEngine.startMatlabAsync` — Start a MATLAB session asynchronously.

You should always terminate the MATLAB session using one of the methods in “Close MATLAB Engine Session” on page 12-15.

Start MATLAB Synchronously

Start MATLAB from Java synchronously.

```
import com.mathworks.engine.*;

public class StartMatlab {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        ...
        eng.close();
    }
}
```

Start MATLAB Asynchronously

Start MATLAB from Java asynchronously. Use the `get` method of the returned `Future` object to wait for the return of the `MatlabEngine` object.

```
import com.mathworks.engine.*;
import java.util.concurrent.Future;

public class StartMatlab {
    public static void main(String[] args) throws Exception {
```

```

    Future<MatlabEngine> engFuture = MatlabEngine.startMatlabAsync();
    //Do work while MATLAB engine starts
    ...
    MatlabEngine eng = engFuture.get();
    ...
    eng.close();
}
}

```

Start Engine with Startup Options

You can specify MATLAB startup options when you start a MATLAB session. For information on MATLAB startup options, see “Commonly Used Startup Options”.

The `MatlabEngine.startMatlab` and `MatlabEngine.startMatlabAsync` methods accept a string array as an input.

Start the engine synchronously with MATLAB startup options.

```

import com.mathworks.engine.*;

public class StartMatlab {
    String[] options = {"-noFigureWindows", "-r", "cd H:"};
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab(options);
        ...
        eng.close();
    }
}

```

Start the engine asynchronously with MATLAB startup options.

```

import com.mathworks.engine.*;
import java.util.concurrent.Future;

public class StartMatlab {
    public static void main(String[] args) throws Exception {
        String[] options = {"-noFigureWindows", "-r", "cd H:"};
        Future<MatlabEngine> engFuture = MatlabEngine.startMatlabAsync(options);
        ...
        MatlabEngine eng = engFuture.get();
        ...
        eng.close();
    }
}

```

Close MATLAB Engine Session

To end the MATLAB engine session, use one of these `MatlabEngine` methods:

Method	Purpose
"close"	If a Java process starts the MATLAB session as a default non-shared session, <code>close()</code> terminates MATLAB. If the MATLAB session is a shared session, <code>close()</code> disconnects MATLAB from this Java process. MATLAB terminates when there are no other connections.
"disconnect", "disconnectAsync"	Disconnect from the current MATLAB session synchronously or asynchronously.
"quit", "quitAsync"	Force the shutdown of the current MATLAB session synchronously or asynchronously.

See Also

`com.mathworks.engine.MatlabEngine`

More About

- "Connect Java to Running MATLAB Session" on page 12-17
- "Build Java Engine Programs" on page 12-3
- "Specify Startup Options"
- "Commonly Used Startup Options"

Connect Java to Running MATLAB Session

In this section...

“Find and Connect to MATLAB” on page 12-17

“Connect to MATLAB Synchronously” on page 12-17

“Connect to MATLAB Asynchronously” on page 12-18

“Specify Name of Shared Session” on page 12-18

Find and Connect to MATLAB

You can connect the Java engine to shared MATLAB sessions that are running on the local machine. To connect to a shared MATLAB session:

- Start MATLAB as a shared engine session, or make a running MATLAB process shared using `matlab.engine.shareEngine`.
- Find the names of the MATLAB shared sessions using the `MatlabEngine.findMatlab` or `MatlabEngine.findMatlabAsync` static method.
- Pass the string containing the name of the shared MATLAB session to the `MatlabEngine.connectMatlab` or `MatlabEngine.connectMatlabAsync` static method. These methods connect the Java engine to the shared session.

If you do not specify the name of a shared MATLAB session when calling `MatlabEngine.connectMatlab` or `MatlabEngine.connectMatlabAsync`, the engine uses the first shared MATLAB session created. If there are no shared MATLAB sessions available, the engine creates a shared MATLAB session and connects to this session.

For a description of these methods, see `com.mathworks.engine.MatlabEngine`

Connect to MATLAB Synchronously

Convert the MATLAB session to a shared session by calling `matlab.engine.shareEngine` from MATLAB.

```
matlab.engine.shareEngine
```

Find the session and connect synchronously from Java.

```
import com.mathworks.engine.*;
```

```
public class javaFindConnect {
    public static void main(String[] args) throws Exception {
        String[] engines = MatlabEngine.findMatlab();
        MatlabEngine eng = MatlabEngine.connectMatlab(engines[0]);
        // Execute command on shared MATLAB session
        eng.eval("plot(1:10); print('myPlot','-djpeg')");
        eng.close();
    }
}
```

Connect to MATLAB Asynchronously

Convert the MATLAB session to a shared session by calling `matlab.engine.shareEngine` from MATLAB.

```
matlab.engine.shareEngine
```

Find the session and connect asynchronously from Java.

```
import com.mathworks.engine.*;
import java.util.concurrent.Future;

public class javaFindConnectAsync {
    public static void main(String[] args) throws Exception {
        Future<String[]> eFuture = MatlabEngine.findMatlabAsync();
        String[] engines = eFuture.get();
        Future<MatlabEngine> engFuture = MatlabEngine.connectMatlabAsync(engines[0]);
        // Work on other thread
        MatlabEngine eng = engFuture.get();
        // Execute command on shared MATLAB session
        Future<Void> vFuture = eng.evalAsync("plot(1:10); print('myPlot','-djpeg')");
        eng.close();
    }
}
```

Specify Name of Shared Session

You can specify the name of the shared MATLAB session when you execute the `matlab.engine.shareEngine` MATLAB function. Doing so eliminates the need to use `MatlabEngine.findMatlab` or `MatlabEngine.findMatlabAsync` to find the name.

For example, start MATLAB and name the shared session `myMatlabEngine`.

```
matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
```

Connect to the named MATLAB session from Java.


```
import com.mathworks.engine.*;

public class javaNameConnect {
    public static void main(String[] args) throws Exception {
        String[] myEngine = {"myMatlabEngine"};
        MatlabEngine eng = MatlabEngine.connectMatlab(myEngine[0]);
        // Execute command on shared MATLAB session
        eng.eval("plot(1:10); print('myPlot','-djpeg')");
        eng.close();
    }
}
```

See Also

Related Examples

- “Start and Close MATLAB Session from Java” on page 12-14

Execute MATLAB Functions from Java

In this section...

“Calling MATLAB Functions” on page 12-20

“Execute Function with Single Returned Argument” on page 12-20

“Execute Function with Multiple Returned Arguments” on page 12-21

“When to Specify Number of Output Arguments” on page 12-21

Calling MATLAB Functions

You can execute MATLAB functions from Java using the `MatlabEngine feval` and `fevalAsync` methods. These methods work like the MATLAB `feval` function. Use `feval` and `fevalAsync` when you want to return the result of the function execution to Java or to pass arguments from Java.

To call a MATLAB function:

- Pass the function name as a string.
- Define the input arguments required by the MATLAB function.
- Specify the number of outputs expect from the MATLAB function (1 is assumed if not specified).
- Define the appropriate returned type for the outputs of the MATLAB function.
- Use writers to redirect output from the MATLAB command window to Java.

You can also use the `MatlabEngine eval` and `evalAsync` methods to evaluate MATLAB expressions. These methods enable you to create variables in the MATLAB workspace, but do not return values.

Execute Function with Single Returned Argument

This example code uses the MATLAB `sqrt` function to find the square root of the elements in an array of doubles. The `feval` method returns a double array containing the results of the `sqrt` function call.

```
import com.mathworks.engine.*;

public class javaFevalFunc{
```

```
public static void main(String[] args) throws Exception{
    MatlabEngine eng = MatlabEngine.startMatlab();
    double[] a = {2.0 ,4.0, 6.0};
    double[] roots = eng.feval("sqrt", a);
    for (double e: roots) {
        System.out.println(e);
    }
    eng.close();
}
```

Execute Function with Multiple Returned Arguments

This example code uses the MATLAB `gcd` function to find the greatest common divisor and Bézout coefficients from the two integer values passed as input arguments. The `feval` method returns an object array containing the results of the `gcd` function call. The returned values are integers.

Because the MATLAB `gcd` function is returning three output arguments, specify the number of returned values as the first argument to the `feval` method.

```
import com.mathworks.engine.*;

public class javaFevalFcnMulti {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        Object[] results = eng.feval(3, "gcd", 40, 60);
        Integer G = (Integer)results[0];
        Integer U = (Integer)results[1];
        Integer V = (Integer)results[2];
        eng.close();
    }
}
```

When to Specify Number of Output Arguments

The `MatlabEngine` `feval` and `fevalAsync` methods enable you to specify the number of output arguments returned by the MATLAB function. By default, the number of output arguments from the MATLAB function is assumed to be 1.

If you want to call a MATLAB function with no outputs or more than one output, specify that number as the first argument passed to `feval` or `fevalAsync`.

For example, this code calls the MATLAB `gcd` function with the three output argument syntax:

```
Object[] results = eng.feval(3, "gcd", 40, 60);
```

MATLAB functions can behave differently depending on the number of outputs requested. Some functions can return no outputs or a specified number of outputs. For example, the MATLAB `pause` function holds execution for a specified number of seconds. However, if you call `pause` with an output argument, the function returns immediately with a status value. Therefore, this code does not cause MATLAB to pause because `feval` requests one output argument.

```
eng.feval("pause", 10);
```

To pause MATLAB execution for the 10 seconds requested, specify the number of outputs as `0`.

```
eng.feval(0, "pause", 10);
```

Note To ensure that a MATLAB function is called with no outputs, specify the number of returned arguments as `0`.

See Also

Related Examples

- “Evaluate MATLAB Statements from Java” on page 12-23

Evaluate MATLAB Statements from Java

In this section...

“Evaluating MATLAB Statements” on page 12-23

“Evaluate Mathematical Function in MATLAB” on page 12-23

Evaluating MATLAB Statements

You can evaluate MATLAB statements from Java using the `MatlabEngine eval` and `evalAsync` methods. These methods are similar to the MATLAB `eval` function. However, the `eval` and `evalAsync` methods do not return the results of evaluating the MATLAB statement.

You can also use the `MatlabEngine feval` and `fevalAsync` methods to call MATLAB functions. These methods enable you to pass variables to the MATLAB workspace and return values to Java.

The input arguments named in the string must exist in the MATLAB workspace. You can assign the results of the evaluation to variables within the statement string. The variable names that you assign in the statement are created in the MATLAB base workspace. MATLAB does not require you to initialize the variables created in the expression.

To return the variables created in the MATLAB workspace, use the `MatlabEngine getVariable` or `getVariableAsync` methods.

Evaluate Mathematical Function in MATLAB

This example code evaluates a mathematical function over a specified domain using two MATLAB statements. The `meshgrid` function creates MATLAB variables `X`, `Y`, and `Z` in the MATLAB workspace. These variables are used by the mathematical expression in the next call to `evalAsync`.

The `MatlabEngine getVariable` method returns the result of the evaluation to Java.

```
import com.mathworks.engine.*;

public class javaEvalFunc {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
```

```
        eng.evalAsync("[X, Y] = meshgrid(-2:0.2:2);");
        eng.evalAsync("Z = X .* exp(-X.^2 - Y.^2);");
        Object[] Z = eng.getVariable("Z");
        eng.close();
    }
}
```

See Also

Related Examples

- “Execute MATLAB Functions from Java” on page 12-20

Pass Variables from Java to MATLAB

In this section...

“Ways to Pass Variables” on page 12-25

“Pass Function Arguments” on page 12-25

“Put Variables in the MATLAB Workspace” on page 12-26

Ways to Pass Variables

You can pass Java variables to MATLAB using these methods:

- Pass the variables as function arguments in calls to the `MatlabEngine` `feval` and `fevalAsync` methods. Variables passed as arguments to function calls are not stored in the MATLAB base workspace.
- Put the variables in the MATLAB base workspace using the `MatlabEngine` `putVariable` and `putVariableAsync` methods.

For information on type conversions, see “Java Data Type Conversions” on page 12-11.

Pass Function Arguments

This example code passes the coefficients of a polynomial, $x^2 - x - 6$, to the MATLAB `roots` function.

- Define a double array `p` to pass as an argument for the MATLAB `roots` function.
- Define a double array `r` to accept the returned values.

```
import com.mathworks.engine.*;

public class javaPassArg{
    public static void main(String[] args) throws Exception{
        MatlabEngine eng = MatlabEngine.startMatlab();
        double[] p = {1.0, -1.0, -6.0};
        double[] r = eng.feval("roots", p);
        for (double e: r) {
            System.out.println(e);
        }
        eng.close();
    }
}
```

```
    }  
}
```

Put Variables in the MATLAB Workspace

This example code puts variables in the MATLAB workspace and uses those variables as arguments in a MATLAB call to the MATLAB `complex` function. The MATLAB `who` command lists the workspace variables.

```
import com.mathworks.engine.*;  
import java.util.Arrays;  
  
public class javaPutVar {  
    public static void main(String[] args) throws Exception {  
        MatlabEngine eng = MatlabEngine.startMatlab();  
        eng.putVariable("x", 7.0);  
        eng.putVariable("y", 3.0);  
        eng.eval("z = complex(x, y);");  
        String[] w = eng.feval("who");  
        System.out.println("MATLAB workspace variables " + Arrays.toString(w));  
        eng.close();  
    }  
}
```

See Also

Related Examples

- “Pass Variables from MATLAB to Java” on page 12-27
- “Base and Function Workspaces”

Pass Variables from MATLAB to Java

Use the `MatlabEngine` `getVariable` or `getVariableAsync` methods to get variables from the MATLAB base workspace. To determine the appropriate mapping of MATLAB type to Java type, see “Java Data Type Conversions” on page 12-11.

Coordinate Conversion

This example code uses the MATLAB `cart2sph` function to convert from Cartesian to spherical coordinates. The `getVariable` method gets the returned spherical coordinate variables from the MATLAB base workspace.

```
import com.mathworks.engine.*;

public class GetPolar {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        eng.eval("[az,el,r] = cart2sph(5, 7, 3);");
        double az = eng.getVariable("az");
        double el = eng.getVariable("el");
        double r = eng.getVariable("r");
        System.out.println("Azimuth: " + az);
        System.out.println("Elevation: " + el);
        System.out.println("Radius " + r);
        eng.close();
    }
}
```

See Also

Related Examples

- “Pass Variables from Java to MATLAB” on page 12-25
- “Java Data Type Conversions” on page 12-11

Using Complex Variables in Java

In this section...

“Complex Variables in MATLAB” on page 12-28

“Get Complex Variables from MATLAB” on page 12-28

“Pass Complex Variable to MATLAB Function” on page 12-29

Complex Variables in MATLAB

MATLAB numeric types can represent complex numbers. The MATLAB Engine API supports complex variables in Java using the `com.mathworks.matlab.types.Complex` class. Using this class, you can:

- Create complex variables in Java and pass these variables to MATLAB.
- Get complex variables from the MATLAB base workspace.

MATLAB always uses double precision values for the real and imaginary parts of complex numbers.

Get Complex Variables from MATLAB

This example code uses MATLAB functions to:

- Find the roots of a polynomial (`roots`)
- Find the complex conjugate of the complex roots (`conj`)
- Find the real result of multiplying the complex number array by its conjugate.

Use the `getVariable` method to return the complex variables to Java.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.Complex;

public class javaGetVar {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        eng.eval("z = roots([1.0, -1.0, 6.0]);");
        eng.eval("zc = conj(z);");
        eng.eval("rat = z.*zc;");
    }
}
```

```

        Complex[] z = eng.getVariable("z");
        Complex[] zc = eng.getVariable("zc");
        double[] rat = eng.getVariable("rat");
        for (Complex e: z) {
            System.out.println(e);
        }
        for (Complex e: zc) {
            System.out.println(e);
        }
        for (double e: rat) {
            System.out.println(e);
        }
        eng.close();
    }
}

```

Pass Complex Variable to MATLAB Function

This example code creates a `com.mathworks.matlab.types.Complex` variable in Java and passes it to the MATLAB `real` function. This function returns the real part of the complex number. The value returned by MATLAB is of type `double` even though the original variable created in Java is an `int`.

```

import com.mathworks.engine.*;
import com.mathworks.matlab.types.Complex;

public class javaComplexVar {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        int r = 8;
        int i = 3;
        Complex c = new Complex(r, i);
        double real = eng.feval("real", c);
        eng.close();
    }
}

```

See Also

Related Examples

- “Java Data Type Conversions” on page 12-11

Using MATLAB Structures in Java

In this section...

“MATLAB Structures” on page 12-31

“Pass Struct to MATLAB Function” on page 12-31

“Get Struct from MATLAB” on page 12-32

MATLAB Structures

MATLAB structures contain data and references it using field names. Each field can contain any type of data. MATLAB code accesses data in a structure using dot notation of the form `structName.fieldName`. The class of a MATLAB structure is `struct`.

The Java `com.mathworks.matlab.types.Struct` class enables you to:

- Create a `Struct` in Java and pass it to MATLAB.
- Create a MATLAB `struct` and return it to Java.

The `com.mathworks.matlab.types.Struct` class implements the `java.util.Map` interface. However, you cannot change the mappings, keys, or values of a `Struct` returned from MATLAB.

Pass Struct to MATLAB Function

The MATLAB `set` function sets the properties of MATLAB graphics objects. To set several properties in one call to `set`, it is convenient to use a MATLAB `struct`. Define this `struct` with field names that match the names of the properties that you want to set. The value referenced by the field is the value assigned the property.

This example code performs the following steps:

- Start MATLAB.
- Pass a double array to the MATLAB `plot` function.
- Return the MATLAB handle object to Java as a `com.mathworks.matlab.types.HandleObject`.
- Create a `com.mathworks.matlab.types.Struct` using property names and values.

- Create a MATLAB graph and display it for 5 seconds.
- Pass the `HandleObject` and the `Struct` to the MATLAB `set` function using `feval`. This function changes the color and line width of the plotted data.
- Export the plot to the jpeg file named `myPlot` and closes the engine connection.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

public class CreateStruct {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        double[] y = {1.0, 2.0, 3.0, 4.0, 5.0};
        HandleObject h = eng.feval("plot", y);
        eng.eval("pause(5)");
        double[] color = {1.0, 0.5, 0.7};
        Struct s = new Struct("Color", color, "LineWidth", 2);
        eng.feval("set", h, s);
        eng.eval("print('myPlot', '-djpeg')");
        eng.close();
    }
}
```

Get Struct from MATLAB

The MATLAB `axes` function creates axes for a graph and returns a handle object reference. The MATLAB `get` function, when called with one output, returns a MATLAB `struct` with the properties of the graphics object.

This example code:

- Creates a MATLAB graphics object and returns the object handle as a `HandleObject`.
- Creates a MATLAB structure containing the properties and values of the graphics object and returns it as a `Struct`.
- Gets the value of the `FontName` property from the `Struct`.
- Attempts to change the value of the `FontName` key, which throws an `UnsupportedOperationException` because the `Struct` is unmodifiable.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

public class GetStruct {
```

```
public static void main(String[] args) throws Exception {
    MatlabEngine eng = MatlabEngine.startMatlab();
    HandleObject h = eng.feval("axes");
    Struct s = eng.feval("get", h);
    Object fontName = s.get("FontName");
    System.out.println("The font name is " + fontName.toString());
    try {
        s.put("FontName", "Times");
    } catch (UnsupportedOperationException e) {
        e.printStackTrace();
    }
    eng.close();
}
```

See Also

Related Examples

- “Java Data Type Conversions” on page 12-11

Pass Java CellStr to MATLAB

In this section...

“MATLAB Cell Arrays” on page 12-34

“Create CellStr” on page 12-34

MATLAB Cell Arrays

MATLAB cell arrays can contain variable-length character vectors in each cell. Some MATLAB functions require cell arrays of character vectors as input arguments. Use the `com.mathworks.matlab.types.CellStr` class to define a cell array of character vectors in Java.

The MATLAB engine converts MATLAB cell arrays of character vectors to Java `String` arrays when passed from MATLAB to Java.

Create CellStr

This example code creates a `MATLAB containers.Map` instance by passing a `CellStr` object and a double array as arguments to the `MATLAB containers.Map` constructor.

Because `containers.Map` is a MATLAB handle class, define the returned type as a `com.mathworks.matlab.types.HandleObject`.

The `containers.Map keys` method returns a MATLAB `cellstr` with the key names. However, the `MatlabEngine feval` method returns a `String` array to Java.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

public class CellArrays {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        CellStr keyCellStr = new CellStr(new String[]{"One", "Two", "Three"});
        double[] valueObject = {1.0, 2.0, 3.0};
        HandleObject myMap = eng.feval("containers.Map", keyCellStr, valueObject);
        String[] keysArray = eng.feval("keys", myMap);
        for (String e: keysArray) {
            System.out.println(e);
        }
        eng.close();
    }
}
```


See Also

Related Examples

- “Java Data Type Conversions” on page 12-11

Using MATLAB Handle Objects in Java

In this section...

“MATLAB Handle Objects” on page 12-36

“Java HandleObject Class” on page 12-36

“Set Graphics Object Properties from Java” on page 12-36

MATLAB Handle Objects

MATLAB handle objects are instances of the `handle` class. Accessing MATLAB handle objects enables you to set the values of public properties on those objects. For example, all MATLAB graphics and user interface objects are handle objects.

Java HandleObject Class

Use the `com.mathworks.matlab.types.HandleObject` class to represent handle objects returned from MATLAB to Java. You can pass the `HandleObject` instance only to the MATLAB session in which it was created. You cannot construct a `HandleObject` in Java.

Set Graphics Object Properties from Java

The MATLAB `plot` function returns the handle objects referencing the lines in the graph. Use these handles with the `set` function to modify the appearance of the graph by changing the properties of the lines.

This example executes the following function calls in MATLAB:

```
% Create a 2-by-3 array of doubles
data = [1,2,3;-1,-2,-3];
% Plot the data and return the line handles
h = plot(data);
% Set the line width to 2 points
set(h,'LineWidth',2);
% Pause for 5 seconds, just to see the result
pause(5)
```

The Java code uses these steps to cause the execution of the MATLAB code as described:

- Create a 2D double array called `data`.
- Cast the `data` array to an `Object` so MATLAB interprets the array as one argument to `plot`.
- Return `HandleObject` array `h` from MATLAB with the line handles.
- Call the MATLAB `set` function to set the `LineWidth` property of the line handles to 2.0. Convert the name of the `LineWidth` property from a `String` to a `char[]` because the `set` function requires property names to be MATLAB `char` arrays.
- Pause for 5 seconds and then close the MATLAB engine.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

public class PassHandleObject {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        double[][] data = {{1.0, 2.0, 3.0}, {-1.0, -2.0, -3.0}};
        HandleObject[] h = eng.feval("plot", (Object) data);
        String lw = ("LineWidth");
        eng.feval(0, "set", h, lw.toCharArray(), 2.0);
        eng.eval("pause(5)");
        eng.close();
    }
}
```

See Also

Related Examples

- “Java Data Type Conversions” on page 12-11

Redirect MATLAB Command Window Output to Java

In this section...

“Output to MATLAB Command Window” on page 12-38

“Redirect MATLAB Output” on page 12-38

“Redirect MATLAB Error Messages to Java” on page 12-39

Output to MATLAB Command Window

MATLAB displays error messages and the output from functions that are not terminated with a semicolon in the MATLAB command window. You can redirect this output to Java using a `java.io.StringWriter`. The `MatlabEngine` `feval`, `fevalAsync`, `eval`, and `evalAsync` methods support the use of output streams to redirect MATLAB output.

Redirect MATLAB Output

The MATLAB `whos` command displays information about the current workspace variables in the MATLAB command window. Use a `StringWriter` to stream this output to Java.

```
import com.mathworks.engine.*;
import java.io.*;

public class RedirectOutput {
    public static void main(String[] args) throws Exception {
        MatlabEngine engine = MatlabEngine.startMatlab();
        // Evaluate expressions that create variables
        eng.evalAsync("[X,Y] = meshgrid(-2:.2:2);");
        eng.evalAsync("Z = X.*exp(-X.^2 - Y.^2);");
        // Get the output of the whos command
        StringWriter writer = new StringWriter();
        eng.eval("whos", writer, null);
        System.out.println(writer.toString());
        writer.close();
        eng.close();
    }
}
```

Redirect MATLAB Error Messages to Java

This example code attempts to evaluate a MATLAB statement that has a syntax error (unbalanced single quotation marks). Entering this statement in MATLAB causes an error:

```
disp('Hello')
```

MATLAB returns this error message in the command window:

```
disp('Hello')
      ^
Error: Character vector is not terminated properly.
```

To redirect this error message to Java, use a `StringWriter` with the `eval` method. Catch the `MatlabSyntaxException` exception thrown by the error and write the MATLAB error message to Java.

```
import com.mathworks.engine.*;
import java.io.*;

public class javaRedirectOut {
    public static void main(String[] args) throws Exception {
        MatlabEngine engine = MatlabEngine.startMatlab();
        StringWriter writer = new StringWriter();
        try {
            eng.eval("disp('Hello')", null, writer);
        } catch (MatlabSyntaxException e) {
            System.out.println("Error redirected to Java: ");
            System.out.println(writer.toString());
        }
        writer.close();
        eng.close();
    }
}
```

See Also

Related Examples

- “Evaluate MATLAB Statements from Java” on page 12-23

Run Simulink Simulation from Java

In this section...
“MATLAB Command to Run Simulation” on page 12-40
“Run vdp Model from Java” on page 12-40

MATLAB Command to Run Simulation

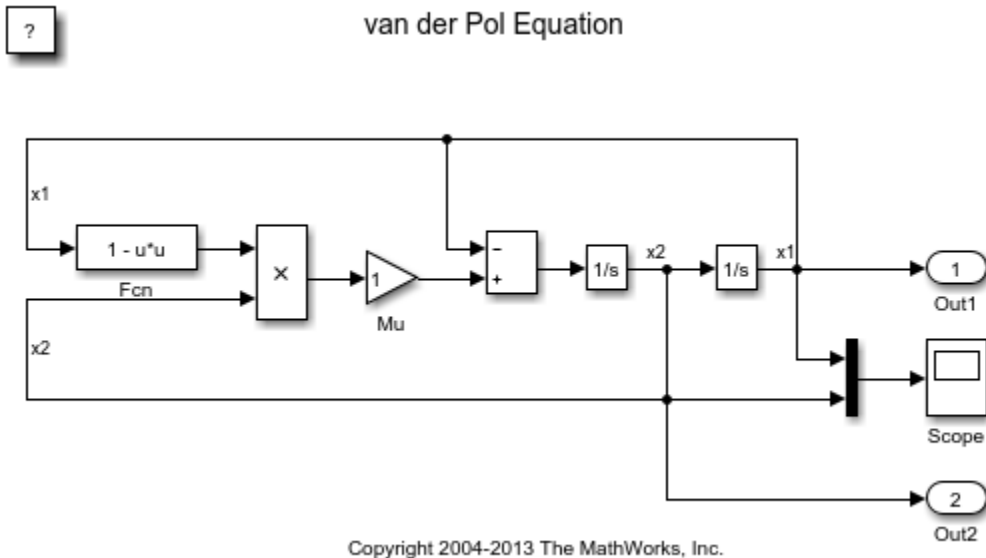
You can run Simulink simulations using the MATLAB Engine API for Java. Here are the basic steps to run a simulation programmatically.

- Create a MATLAB engine object and start a MATLAB session.
- Load the Simulink model in MATLAB (`load_system`).
- Run the simulation with specific simulation parameters (`sim`).
- Access the results of the simulation using methods of the returned `Simulink.SimulationOutput` object.

For information on running simulations programmatically from MATLAB, see “Run Individual Simulations” (Simulink).

Run vdp Model from Java

The Simulink `vdp` block diagram simulates the van der Pol equation, which is a second order differential equation. The model solves the equations using the initial conditions and configuration parameters defined by the model.



MATLAB Code to Run Simulation

This MATLAB code shows the commands to run the simulation programmatically. The `Simulink.SimulationOutput` object `get` method returns the results and time vector.

```
mdl = 'vdp';
load_system(mdl);
simOut = sim(mdl, 'SaveOutput', 'on', ...
    'OutputSaveName', 'yOut', ...
    'SaveTime', 'on', ...
    'TimeSaveName', 'tOut');
y = simOut.get('yOut');
t = simOut.get('tOut');
```

Graph the Data

This MATLAB code creates a graph of the simulation output and exports the graph to a JPEG image file.

```
plot(t,y)
print('vdpPlot', '-djpeg')
```

Java Code to Run Simulation

This Java code runs the Simulink `vdp` model simulation and returns the results to Java. The implementation performs these operations:

- Creates a MATLAB engine object and start a MATLAB session.
- Calls the MATLAB `load_system` command to start Simulink and load the `vdp` model asynchronously. Poll the task until the `Future` returns.
- Calls the MATLAB `sim` command to set simulation parameters and run the simulation. Poll the task until the `Future` returns.
- Captures the result of the simulation. The output of the `sim` function is a MATLAB `Simulink.SimulationOutput` object, which is created in the MATLAB base workspace.

The engine API does not support this type of object. Therefore, this example uses the object `get` method to access the simulation data in the MATLAB workspace.

- Creates a graphic of the simulation data and exports this graph to a JPEG file.
- Returns the simulation results and time vector to Java as `double` arrays.

```
import com.mathworks.engine.*;
import java.util.concurrent.Future;
import java.util.Arrays;

public class RunSimulation {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        Future<Void> fLoad = eng.evalAsync("load_system('vdp')");
        while (!fLoad.isDone()){
            System.out.println("Loading Simulink model...");
            Thread.sleep(10000);
        }
        Future<Void> fSim = eng.evalAsync("simOut = sim('vdp','SaveOutput'," +
            "'on','OutputSaveName','yOut'," +
            "'SaveTime','on','TimeSaveName','tOut');");
        while (!fSim.isDone()) {
            System.out.println("Running Simulation...");
            Thread.sleep(10000);
        }
        // Get simulation data
        eng.eval("y = simOut.get('yOut');");
        eng.eval("t = simOut.get('tOut');");
        // Graph results and create image file
        eng.eval("plot(t,y)");
        eng.eval("print('vdpPlot','-djpeg')");
        // Return results to Java
        double[][] y = eng.getVariable("y");
        double[] t = eng.getVariable("t");
    }
}
```



```
        // Display results
        System.out.println("Simulation result " + Arrays.deepToString(y));
        System.out.println("Time vector " + Arrays.toString(t));
        eng.close();
    }
}
```

See Also

Related Examples

- “Evaluate MATLAB Statements from Java” on page 12-23
- “Pass Variables from Java to MATLAB” on page 12-25
- “Pass Variables from MATLAB to Java” on page 12-27

MATLAB Engine API Exceptions

In this section...
<code>"com.mathworks.engine.EngineException"</code> on page 12-44
<code>"com.mathworks.engine.UnsupportedTypeException"</code> on page 12-44
<code>"com.mathworks.engine.MatlabExecutionException"</code> on page 12-44
<code>"com.mathworks.engine.MatlabSyntaxException"</code> on page 12-44

com.mathworks.engine.EngineException

The `com.mathworks.engine.EngineException` occurs if MATLAB fails to start, fails to connect, terminates, or disconnects. Use the `getMessage` method to get a detailed message string returned by this exception.

com.mathworks.engine.UnsupportedTypeException

The `com.mathworks.engine.UnsupportedTypeException` occurs if you use an unsupported data type in either the input or output of a MATLAB function. Use the `getMessage` method to get a detailed message string returned by this exception.

com.mathworks.engine.MatlabExecutionException

The `com.mathworks.engine.MatlabExecutionException` occurs if there is a runtime error in a MATLAB command. This class inherits from `java.util.ExecutionException`.

com.mathworks.engine.MatlabSyntaxException

The `com.mathworks.engine.MatlabSyntaxException` occurs if there is a syntax error in a MATLAB command. This class inherits from `java.util.ExecutionException`.

See Also

Related Examples

- “Java Engine API Summary” on page 12-8

Pass Java Array Arguments to MATLAB

In this section...

“2-D Array Arguments for MATLAB Functions” on page 12-46

“Multidimensional Arrays” on page 12-46

2-D Array Arguments for MATLAB Functions

Some MATLAB functions accept a 2-D array as a single input argument and use the columns of the array separately. By default, if you pass a 2-D array to a MATLAB from Java, the array is split into separate arguments along the second dimension. To prevent this issue, cast the 2-D array to Object:

```
double[][] data = {{1.0, 2.0, 3.0}, {-1.0, -2.0, -3.0}};
HandleObject[] h = eng.feval("plot", (Object) data);
```

Multidimensional Arrays

MATLAB and Java use different representations to display multidimensional arrays. However, indexing expressions produce the same results. For example, this example code defines an array with three dimensions in MATLAB. The array variable is then passed to Java and the results of indexed references are compared.

```
import com.mathworks.engine.*;
import java.io.StringWriter;
import java.util.Arrays;

public class ArrayIndexing {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        StringWriter writer = new StringWriter();
        eng.eval("A(1:2,1:3,1) = [1,2,3;4,5,6];");
        eng.eval("A(1:2,1:3,2) = [7,8,9;10,11,12]");
        double[][][] A = eng.getVariable("A");
        System.out.println("In Java: \n"+ Arrays.deepToString(A));
        eng.eval("A(1,1,:)",writer,null);
        System.out.println("A(1,1,:) " + writer.toString());
        System.out.println("Java [0][0][0] " + A[0][0][0]);
        System.out.println("Java [0][0][1] " + A[0][0][1]);
    }
}
```

Here is the program output showing how MATLAB and Java display the arrays. In MATLAB:

```
A(:,:,1) =
```

```
    1    2    3
    4    5    6
```

```
A(:,:,2) =
```

```
    7    8    9
   10   11   12
```

In Java:

```
[[[1.0, 7.0], [2.0, 8.0], [3.0, 9.0]], [[4.0, 10.0], [5.0, 11.0], [6.0, 12.0]]]
```

Here are the results showing indexed reference to the first element in each outer dimension:

```
A(1,1,:)
```

```
ans(:,:,1) =
```

```
    1
```

```
ans(:,:,2) =
```

```
    7
```

```
Java [0][0][0] 1.0
```

```
Java [0][0][1] 7.0
```

In MATLAB and Java, the results of the indexed expression are the same.

See Also

Related Examples

- “Multidimensional Arrays”

Incorrect Java Data Types

In this section...

“Java String to MATLAB Character Vector” on page 12-48

“Setting Graphics Object Properties from Java” on page 12-49

“Java Integer to MATLAB double” on page 12-49

Java String to MATLAB Character Vector

Struct of Character Vectors

Some MATLAB functions accept a `struct` of name-value pairs as input arguments. The MATLAB Engine API for Java provides the `com.mathworks.matlab.engine.Struct` class to create this data structure in Java and pass it to MATLAB, where it is converted to a MATLAB `struct`.

Some MATLAB functions that accept `struct` input require field values to be MATLAB character vectors (`char`) instead of MATLAB strings (`string`). To create a Java `Struct` with the correct type of values, convert from `String` to `char` array before passing the variable to MATLAB.

You can use the `toCharArray` method for the conversion:

```
char[] on = "on".toCharArray();
char[] yOut = "yOut".toCharArray();
char[] tOut = "tOut".toCharArray();
Struct simParam = new Struct("SaveOutput", on, "OutputSaveName",
    yOut, "SaveTime", on, "TimeSaveName", tOut);
```

String Argument to Character Vector

When MATLAB functions require `char` inputs, you can convert the Java `String` in the function call passed to MATLAB. For example, the MATLAB `eval` function requires `char` input:

```
double result = engine.feval("eval", "3+5");
Undefined function 'eval' for input arguments of type 'string'..
```

Passing a `char` array works correctly.

```
double result = engine.feval("eval", "3+5".toCharArray());
```

Setting Graphics Object Properties from Java

You can set the values of MATLAB graphics object properties using the handle of the object. Pass the property names and property values as Java char arrays when passing to MATLAB functions.

```
double[][] data = {{1.0, 2.0, 3.0}, {-1.0, -2.0, -3.0}};
HandleObject[] h = eng.feval("plot", (Object)data);
String property = ("HitTest");
String value = ("off");
eng.feval(0,"set", h, property.toCharArray(), value.toCharArray());
```

Java Integer to MATLAB double

Some MATLAB functions, such as `sqrt` restrict the input to double or single precision values. The MATLAB engine converts Java integers to MATLAB `int32` values. For MATLAB functions that do not accept integer values, ensure that you pass appropriate numeric values.

```
double result = engine.feval("sqrt", 4);
Undefined function 'sqrt' for input arguments of type 'int32'.
```

Passing a double works correctly.

```
double result = engine.feval("sqrt", 4.0);
```

See Also

Related Examples

- “Java Data Type Conversions” on page 12-11

Incorrect Number of Output Arguments

By default, the `feval` and `fevalAsync` methods request one output argument when calling MATLAB functions. Some MATLAB functions behave differently depending on the number of output arguments requested. Use the first input argument to specify the number of required output arguments as follows:

- If you want to call the function with no outputs, specify the first argument as `0`
- If you want to call the function with more than one output, specify the exact number.

For example, the MATLAB `disp` function does not return an output argument. This call to `disp` requires one output argument:

```
engine.feval("disp", 100);  
Error using disp  
Too many output arguments.
```

You must specify the number of output arguments explicitly as `0`:

```
engine.feval(0, "disp", 100);
```

See Also

Related Examples

- “Execute MATLAB Functions from Java” on page 12-20

MATLAB Engine for Python Topics

- “Get Started with MATLAB Engine API for Python” on page 13-2
- “Install MATLAB Engine API for Python” on page 13-5
- “Install MATLAB Engine API for Python in Nondefault Locations” on page 13-7
- “Start and Stop MATLAB Engine for Python” on page 13-9
- “Connect Python to Running MATLAB Session” on page 13-11
- “Call MATLAB Functions from Python” on page 13-14
- “Call MATLAB Functions Asynchronously from Python” on page 13-16
- “Call User Scripts and Functions from Python” on page 13-17
- “Redirect Standard Output and Error to Python” on page 13-19
- “Use MATLAB Handle Objects in Python” on page 13-20
- “Use MATLAB Engine Workspace in Python” on page 13-22
- “Pass Data to MATLAB from Python” on page 13-23
- “Handle Data Returned from MATLAB to Python” on page 13-25
- “MATLAB Arrays as Python Variables” on page 13-28
- “Use MATLAB Arrays in Python” on page 13-34
- “Sort and Plot MATLAB Data from Python” on page 13-36
- “Get Help for MATLAB Functions from Python” on page 13-40
- “Default Numeric Types in MATLAB and Python” on page 13-42
- “System Requirements for MATLAB Engine API for Python” on page 13-44
- “Limitations to MATLAB Engine API for Python” on page 13-46
- “Troubleshoot MATLAB Errors in Python” on page 13-47

Get Started with MATLAB Engine API for Python

The MATLAB Engine API for Python provides a Python package named `matlab` that enables you to call MATLAB functions from Python. You install the package once, and then you can call the engine in your current or future Python sessions. For help on installing or starting the engine, refer to:

- “Install MATLAB Engine API for Python” on page 13-5
- “Start and Stop MATLAB Engine for Python” on page 13-9

The `matlab` package contains the following:

- The MATLAB Engine API for Python
- A set of MATLAB array classes in Python (see “MATLAB Arrays as Python Variables” on page 13-28)

The engine provides functions to call MATLAB, and the array classes provide functions to create MATLAB arrays as Python objects. You can create an engine and call MATLAB functions with `matlab.engine`. You can create MATLAB arrays in Python by calling constructors of an array type (for example, `matlab.double` to create an array of doubles). MATLAB arrays can be input arguments to MATLAB functions called with the engine.

The table shows the structure of the `matlab` package.

Package	Function or Class	Description
<code>matlab.engine</code>	<code>start_matlab()</code>	Python function to create a <code>MatlabEngine</code> object, and attach it to a new MATLAB process
<code>matlab.engine</code>	<code>MatlabEngine</code>	Python class to provide methods for calling MATLAB functions

Package	Function or Class	Description
matlab.engine	FutureResult	Python class to hold results from a MATLAB function called asynchronously
matlab	double	Python class to hold array of MATLAB type double
matlab	single	Python class to hold array of MATLAB type single
matlab	int8	Python class to hold array of MATLAB type int8
matlab	int16	Python class to hold array of MATLAB type int16
matlab	int32	Python class to hold array of MATLAB type int32
matlab	int64	Python class to hold array of MATLAB type int64
matlab	uint8	Python class to hold array of MATLAB type uint8
matlab	uint16	Python class to hold array of MATLAB type uint16
matlab	uint32	Python class to hold array of MATLAB type uint32

Package	Function or Class	Description
matlab	uint64	Python class to hold array of MATLAB type uint64
matlab	logical	Python class to hold array of MATLAB type logical
matlab	object	Python class to hold a handle to a MATLAB object

See Also

More About

- “System Requirements for MATLAB Engine API for Python” on page 13-44

Install MATLAB Engine API for Python

To start the MATLAB engine within a Python session, you first must install the engine API as a Python package. MATLAB provides a standard Python `setup.py` file for building and installing the engine using the `distutils` module. You can use the same `setup.py` commands to build and install the engine on Windows, Mac, or Linux systems.

Before you install, verify your Python and MATLAB configurations.

- Check that your system has a supported version of Python and MATLAB R2014b or later. To check that Python is installed on your system, run Python at the operating system prompt.
- Add the folder that contains the Python interpreter to your path, if it is not already there.
- Find the path to the MATLAB folder. Start MATLAB and type `matlabroot` in the command window. Copy the path returned by `matlabroot`.

To install the engine API, choose one of the following.

- At a Windows operating system prompt —

```
cd "matlabroot\extern\engines\python"  
python setup.py install
```

You might need administrator privileges to execute these commands.

- At a macOS or Linux operating system prompt —

```
cd "matlabroot/extern/engines/python"  
python setup.py install
```

You might need administrator privileges to execute these commands.

- At the MATLAB command prompt —

```
cd (fullfile(matlabroot, 'extern', 'engines', 'python'))  
system('python setup.py install')
```

- Use one of the nondefault options described in “Install MATLAB Engine API for Python in Nondefault Locations” on page 13-7.

Note You must call this `python` command in the specified folder.

Start Python, import the module, and start the MATLAB engine:

```
import matlab.engine  
eng = matlab.engine.start_matlab()
```

See Also

More About

- “System Requirements for MATLAB Engine API for Python” on page 13-44
- “Install Supported Python Implementation” on page 20-3
- “Install MATLAB Engine API for Python in Nondefault Locations” on page 13-7

External Websites

- Python 2.7 Documentation — Installing Python Modules

Install MATLAB Engine API for Python in Nondefault Locations

In this section...

“Build or Install in Nondefault Folders” on page 13-7

“Install Engine in Your Home Folder” on page 13-8

Build or Install in Nondefault Folders

By default, the installer builds the engine API for Python in the *matlabroot*\extern\engines\python folder. The installer installs the engine in the default Python folder. If you do not have write permission for these folders, then select one of the following nondefault options. If you install in another folder, then update the PYTHONPATH variable to the location of that folder.

Options for building and installing the engine API follow, along with the commands to enter at the operating system prompt.

Build in Nondefault Folder and Install in Default Folder

If you do not have write permission to build the engine in the MATLAB folder, then use a nondefault folder, *builddir*.

```
cd "matlabroot\extern\engines\python"  
python setup.py build --build-base="builddir" install
```

Build in Default Folder and Install in Nondefault Folder

If you do not have write permission to install the engine in the default Python folder, then use a nondefault folder, *installdir*.

```
cd "matlabroot\extern\engines\python"  
python setup.py install --prefix="installdir"
```

To include *installdir* in the search path for Python packages, add *installdir* to the PYTHONPATH environment variable.

Build and Install in Nondefault Folders

If you do not have write permission for both the MATLAB folder and the default Python folder, then you can specify nondefault folders. Use *builddir* for the build folder and *installdir* for the install folder.

```
cd "matlabroot\extern\engines\python"  
python setup.py build --build-base="builddir" install --prefix="installdir"
```

Install Engine in Your Home Folder

To install the engine API for your use only, use the `--user` option to install in your home folder.

```
cd "matlabroot\extern\engines\python"  
python setup.py install --user
```

When you install with `--user`, you do not need to add your home folder to PYTHONPATH.

See Also

Related Examples

- “Install MATLAB Engine API for Python” on page 13-5

More About

- “System Requirements for MATLAB Engine API for Python” on page 13-44

Start and Stop MATLAB Engine for Python

In this section...

“Start MATLAB Engine for Python” on page 13-9

“Run Multiple Engines” on page 13-9

“Stop Engine” on page 13-9

“Start Engine with Startup Options” on page 13-10

“Start Engine Asynchronously” on page 13-10

Start MATLAB Engine for Python

- Start Python at the operating system prompt.
- Import the `matlab.engine` package into your Python session.
- Start a new MATLAB process by calling `start_matlab`. The `start_matlab` function returns a Python object, `eng`, which enables you to pass data and call functions executed by MATLAB.

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

Run Multiple Engines

Start each engine separately. Each engine starts and communicates with its own MATLAB process.

```
eng1 = matlab.engine.start_matlab()
eng2 = matlab.engine.start_matlab()
```

Stop Engine

Call either the `exit` or the `quit` function.

```
eng.quit()
```

If you exit Python with an engine still running, then Python automatically stops the engine and its MATLAB process.

Start Engine with Startup Options

Start the engine and pass the options as an input argument string to `matlab.engine.start_matlab`. For example, start MATLAB with the desktop.

```
eng = matlab.engine.start_matlab("-desktop")
```

You can define multiple startup options with a single string. For example, start the desktop and set the numeric display format to `short`.

```
eng = matlab.engine.start_matlab("-desktop -r 'format short'")
```

You also can start the desktop after you start the engine.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.desktop(nargout=0)
```

Start Engine Asynchronously

Start the engine asynchronously. While MATLAB starts, you can enter commands at the Python command line.

```
import matlab.engine
future = matlab.engine.start_matlab(background=True)
```

Create the MATLAB instance so you can perform computations in MATLAB.

```
eng = future.result()
```

See Also

`matlab.engine.start_matlab`

More About

- “Specify Startup Options”
- “Commonly Used Startup Options”

Connect Python to Running MATLAB Session

You can connect the MATLAB Engine for Python to a shared MATLAB session that is already running on your local machine. You also can connect to multiple shared MATLAB sessions from a single Python session. You can share a MATLAB session at any time during the session, or at start with a startup option.

Connect to Shared MATLAB Session

First, convert your MATLAB session to a shared session. From MATLAB call `matlab.engine.shareEngine`.

```
matlab.engine.shareEngine
```

Start Python at the operating system prompt. To connect to the shared MATLAB session, call `matlab.engine.connect_matlab` from Python. You can call any MATLAB function from Python.

```
import matlab.engine
eng = matlab.engine.connect_matlab()
eng.sqrt(4.0)
```

```
2.0
```

You can connect to a shared session by name. To find the name of a shared session, call `matlab.engine.find_matlab` from Python.

```
matlab.engine.find_matlab()
```

```
('MATLAB_13232',)
```

`matlab.engine.find_matlab` returns a tuple with the names of all shared MATLAB sessions on your local machine. In this example `matlab.engine.shareEngine` gave the shared session the default name `MATLAB_13232`, where 13232 is the ID of the MATLAB process. The operating system gives the MATLAB session a different process ID whenever you start MATLAB.

Connect to the MATLAB session by name.

```
eng.quit()
newEngine = matlab.engine.connect_matlab('MATLAB_13232')
```

If you do not specify the name of a shared session, `matlab.engine.connect_matlab` connects to the first session named in the tuple returned by `matlab.engine.find_matlab`.

Connect Asynchronously to Shared MATLAB Session

From MATLAB, convert your MATLAB session to a shared session.

```
matlab.engine.shareEngine
```

Start Python at the operating system prompt. Connect asynchronously to the shared MATLAB session.

```
import matlab.engine
future = matlab.engine.connect_matlab(background=True)
eng = future.result()
```

Call a MATLAB function from Python.

```
eng.sqrt(4.0)
```

```
2.0
```

Connect to Multiple Shared MATLAB Sessions

You can connect to multiple shared MATLAB sessions from Python.

Start a second MATLAB session. From MATLAB call `matlab.engine.shareEngine`. Give a name to the second shared session. The name must be a valid MATLAB variable name. For information on valid variable names, see “Variable Names”.

```
matlab.engine.shareEngine('MATLABEngine2')
```

From Python, find all shared MATLAB sessions.

```
import matlab.engine
matlab.engine.find_matlab()

('MATLAB_13232', 'MATLABEngine2')
```

To connect to the shared MATLAB sessions, call `matlab.engine.connect_matlab` from Python.

```
eng1 = matlab.engine.connect_matlab('MATLAB_13232')  
eng2 = matlab.engine.connect_matlab('MATLABEngine2')
```

Start Shared MATLAB Sessions with Startup Options

By default MATLAB sessions are not shared. However, you can start MATLAB as a shared session with a startup option.

Start shared MATLAB sessions at the operating system prompt.

```
matlab -r "matlab.engine.shareEngine"  
matlab -r "matlab.engine.shareEngine('MATLABEngine3')"
```

You can start a session with a default name, or give a name enclosed in single quotes.

See Also

```
matlab.engine.connect_matlab | matlab.engine.engineName |  
matlab.engine.find_matlab | matlab.engine.isEngineShared |  
matlab.engine.shareEngine
```

More About

- “Specify Startup Options”
- “Commonly Used Startup Options”

Call MATLAB Functions from Python

In this section...
“Return Output Argument from MATLAB Function” on page 13-14
“Return Multiple Output Arguments from MATLAB Function” on page 13-14
“Return No Output Arguments from MATLAB Function” on page 13-15
“Stop Execution of Function” on page 13-15

Return Output Argument from MATLAB Function

You can call any MATLAB function directly and return the results to Python. For example, to determine if a number is prime, use the engine to call the `isprime` function.

```
import matlab.engine
eng = matlab.engine.start_matlab()
tf = eng.isprime(37)
print(tf)
```

```
True
```

Return Multiple Output Arguments from MATLAB Function

When you call a function with the engine, by default the engine returns a single output argument. If you know that the function can return multiple arguments, use the `nargout` argument to specify the number of output arguments.

To determine the greatest common denominator of two numbers, use the `gcd` function. Set `nargout` to return the three output arguments from `gcd`.

```
import matlab.engine
eng = matlab.engine.start_matlab()
t = eng.gcd(100.0, 80.0, nargout=3)
print(t)
```

```
(20.0, 1.0, -1.0)
```

Return No Output Arguments from MATLAB Function

Some MATLAB functions return no output arguments. If the function returns no arguments, set `nargout` to 0.

Open the MATLAB Help browser from Python.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.doc(nargout=0)
```

The MATLAB `doc` function opens the browser, but does not return output arguments. If you do not specify `nargout=0`, the engine raises an error.

Stop Execution of Function

To stop execution of a MATLAB function press **Ctrl+C**. Control returns to Python.

See Also

`matlab.engine.FutureResult` | `matlab.engine.MatlabEngine`

Related Examples

- “Call MATLAB Functions Asynchronously from Python” on page 13-16
- “Call User Scripts and Functions from Python” on page 13-17
- “Use MATLAB Arrays in Python” on page 13-34
- “Sort and Plot MATLAB Data from Python” on page 13-36

Call MATLAB Functions Asynchronously from Python

This example shows how to call the MATLAB `sqrt` function asynchronously from Python and retrieve the square root later.

The engine calls MATLAB functions synchronously by default. Control returns to Python only when the MATLAB function finishes. But the engine also can call functions asynchronously. Control immediately returns to Python while MATLAB is still executing the function. The engine stores the result in a Python variable that can be inspected after the function finishes.

Use the `background` argument to call a MATLAB function asynchronously.

```
import matlab.engine
eng = matlab.engine.start_matlab()
future = eng.sqrt(4.0,background=True)
ret = future.result()
print(ret)
```

```
2.0
```

Use the `done` method to check if an asynchronous call finished.

```
tf = future.done()
print(tf)
```

```
True
```

To stop execution of the function before it finishes, call `future.cancel()`.

See Also

`matlab.engine.FutureResult` | `matlab.engine.MatlabEngine`

Related Examples

- “Call MATLAB Functions from Python” on page 13-14
- “Call User Scripts and Functions from Python” on page 13-17

Call User Scripts and Functions from Python

This example shows how to call a MATLAB script to compute the area of a triangle from Python.

In your current folder, create a MATLAB script in a file named `triarea.m`.

```
b = 5;  
h = 3;  
a = 0.5*(b.* h)
```

After you save the file, start Python and call the script.

```
import matlab.engine  
eng = matlab.engine.start_matlab()  
eng.triarea(nargout=0)
```

```
a =
```

```
7.5000
```

Specify `nargout=0`. Although the script prints output, it returns no output arguments to Python.

Convert the script to a function and call the function from the engine. To edit the file, open the MATLAB editor.

```
eng.edit('triarea',nargout=0)
```

Delete the three statements. Then add a function declaration and save the file.

```
function a = triarea(b,h)  
a = 0.5*(b.* h);
```

Call the new `triarea` function from the engine.

```
ret = eng.triarea(1.0,5.0)  
print(ret)
```

```
2.5
```

The `triarea` function returns only one output argument, so there is no need to specify `nargout`.

See Also

`matlab.engine.FutureResult` | `matlab.engine.MatlabEngine`

Related Examples

- “Call MATLAB Functions from Python” on page 13-14

Redirect Standard Output and Error to Python

This example shows how to redirect standard output and standard error from a MATLAB function to Python StringIO objects.

In Python 2.7, use the StringIO module to create StringIO objects. To capture a warning message from dec2hex, specify stdout and stderr.

```
import matlab.engine
eng = matlab.engine.start_matlab()
import StringIO
out = StringIO.StringIO()
err = StringIO.StringIO()
ret = eng.dec2hex(2**60, stdout=out, stderr=err)
print(out.getvalue())
```

Warning: At least one of the input numbers is larger than the largest integer-valued floating-point number (2⁵²). Results may be unpredictable.

In Python 3.x, use the io module to create StringIO objects.

```
import matlab.engine
eng = matlab.engine.start_matlab()
import io
out = io.StringIO()
err = io.StringIO()
ret = eng.dec2base(2**60, 16, stdout=out, stderr=err)
```

dec2base raises an exception when an input argument is greater than 2⁵². Display the error message captured in err.

```
print(err.getvalue())
```

```
Error using dec2base (line 22)
First argument must be an array of integers, 0 <= D <= 2^52.
```

See Also

matlab.engine.FutureResult | matlab.engine.MatlabEngine

Related Examples

- “Call MATLAB Functions from Python” on page 13-14

Use MATLAB Handle Objects in Python

This example shows how to create an object from a MATLAB handle class and call its methods in Python.

In your current folder, create a MATLAB handle class in a file named `Triangle.m`.

```
classdef Triangle < handle
    properties (SetAccess = private)
        Base = 0;
        Height = 0;
    end

    methods
        function TR = Triangle(b,h)
            TR.Base = b;
            TR.Height = h;
        end

        function a = area(TR)
            a = 0.5 .* TR.Base .* TR.Height;
        end

        function setBase(TR,b)
            TR.Base = b;
        end

        function setHeight(TR,h)
            TR.Height = h;
        end
    end
end
```

Start Python. Create a `Triangle` handle object and call its `area` method with the engine. Pass the handle object as the first positional argument.

```
import matlab.engine
eng = matlab.engine.start_matlab()
tr = eng.Triangle(5.0,3.0)
a = eng.area(tr)
print(a)
```

7.5

Copy `tr` to the MATLAB workspace. You can use `eval` to access the properties of a handle object from the workspace.

```
eng.workspace["wtr"] = tr
b = eng.eval("wtr.Base")
print(b)
```

5.0

Change the height with the `setHeight` method. If your MATLAB handle class defines `get` and `set` methods for properties, you can access properties without using the MATLAB workspace.

```
eng.setHeight(tr,8.0,nargout=0)
a = eng.area(tr)
print(a)
```

20.0

Note Triangle class object `tr`, is a handle to the object, not a copy of the object. If you create `tr` in a function, it is only valid within the scope of the function.

See Also

`matlab.engine.FutureResult` | `matlab.engine.MatlabEngine`

Related Examples

- “Call MATLAB Functions from Python” on page 13-14

Use MATLAB Engine Workspace in Python

This example shows how to add variables to the MATLAB engine workspace in Python.

When you start the engine, it provides an interface to a collection of all MATLAB variables. This collection, named `workspace`, is implemented as a Python dictionary that is attached to the engine. The name of each MATLAB variable becomes a key in the `workspace` dictionary. The keys in `workspace` must be valid MATLAB identifiers (e.g., you cannot use numbers as keys). You can add variables to the engine workspace in Python, and then you can use the variables in MATLAB functions.

Add a variable to the engine workspace.

```
import matlab.engine
eng = matlab.engine.start_matlab()
x = 4.0
eng.workspace['y'] = x
a = eng.eval('sqrt(y)')
print(a)

2.0
```

In this example, `x` exists only as a Python variable. Its value is assigned to a new entry in the engine workspace, called `y`, creating a MATLAB variable. You can then call the MATLAB `eval` function to execute the `sqrt(y)` statement in MATLAB and return the output value, `2.0`, to Python.

See Also

`matlab.engine.FutureResult` | `matlab.engine.MatlabEngine`

Related Examples

- “Call MATLAB Functions from Python” on page 13-14
- “Sort and Plot MATLAB Data from Python” on page 13-36

Pass Data to MATLAB from Python

In this section...

“Python Type to MATLAB Scalar Type Mapping” on page 13-23

“Python Container to MATLAB Array Type Mapping” on page 13-24

“Unsupported Python Types” on page 13-24

Python Type to MATLAB Scalar Type Mapping

When you pass Python data as input arguments to MATLAB functions, the MATLAB Engine for Python converts the data into equivalent MATLAB data types.

Python Input Argument Type — Scalar Values Only	Resulting MATLAB Data Type
float	double
complex	Complex double
int	int64
long (Python 2.7 only)	int64
float(nan)	NaN
float(inf)	Inf
bool	logical
str	char
unicode (Python 2.7 only)	char
dict	Structure if all keys are strings not supported otherwise

Python Container to MATLAB Array Type Mapping

Python Input Argument Type — Container	Resulting MATLAB Data Type
matlab numeric array object (see “MATLAB Arrays as Python Variables” on page 13-28)	Numeric array
bytearray	uint8 array
bytes (Python 3.x) bytes (Python 2.7)	uint8 array char array
list	Cell array
set	Cell array
tuple	Cell array

Unsupported Python Types

The MATLAB Engine API does not support these Python types.

- `array.array` (use MATLAB numeric array objects instead; see “MATLAB Arrays as Python Variables” on page 13-28)
- `None`
- `module.type` object

See Also

More About

- “Handle Data Returned from MATLAB to Python” on page 13-25
- “Default Numeric Types in MATLAB and Python” on page 13-42
- “MATLAB Arrays as Python Variables” on page 13-28

Handle Data Returned from MATLAB to Python

In this section...

“MATLAB Scalar Type to Python Type Mapping” on page 13-25

“MATLAB Array Type to Python Type Mapping” on page 13-26

“Unsupported MATLAB Types” on page 13-27

MATLAB Scalar Type to Python Type Mapping

When MATLAB functions return output arguments, the MATLAB Engine API for Python converts the data into equivalent Python data types.

MATLAB Output Argument Type — Scalar Values Only	Resulting Python Data Type
double	float
single	float
Complex (any numeric type)	complex
int8	int
uint8	int
int16	int
uint16	int
int32	int
uint32	int (Python 3.x) long (Python 2.7)
int64	int (Python 3.x) long (Python 2.7)
uint64	int (Python 3.x) long (Python 2.7)
NaN	float(nan)
Inf	float(inf)
logical	bool
string	string

MATLAB Output Argument Type – Scalar Values Only	Resulting Python Data Type
<missing> value in string	None
char returned to Python 3.x	str
char returned to Python 2.7	str (when MATLAB char value is less than or equal to 127) unicode (when MATLAB char value is greater than 127)
Structure	dict
MATLAB handle object (such as the <code>containers.Map</code> type)	matlab.object MATLAB returns a reference to a <code>matlab.object</code> , not the object itself. You cannot pass a <code>matlab.object</code> between MATLAB sessions.
MATLAB value object (such as the <code>categorical</code> type)	Opaque object. You can pass a value object to a MATLAB function but you cannot create or modify it.

MATLAB Array Type to Python Type Mapping

MATLAB Output Argument Type – Array	Resulting Python Data Type
Numeric array	matlab numeric array object (see “MATLAB Arrays as Python Variables” on page 13-28)
string vector	list of string
char array (1-by-N, N-by-1) returned to Python 3.x	str
char array (1-by-N, N-by-1) returned to Python 2.7	str (when MATLAB char array has values less than or equal to 127) unicode (when MATLAB char array has any value greater than 127)
Row or column cell array	list

Unsupported MATLAB Types

The MATLAB Engine API for Python does not support these MATLAB data types.

- char array (M-by-N)
- Cell array (M-by-N)
- Sparse array
- Structure array
- Non-MATLAB objects (such as Java objects)

See Also

More About

- “Pass Data to MATLAB from Python” on page 13-23

MATLAB Arrays as Python Variables

In this section...

“Create MATLAB Arrays in Python” on page 13-28

“MATLAB Array Attributes and Methods in Python” on page 13-30

“Multidimensional MATLAB Arrays in Python” on page 13-30

“Index Into MATLAB Arrays in Python” on page 13-30

“Slice MATLAB Arrays in Python” on page 13-31

“Reshape MATLAB Arrays in Python” on page 13-32

The `matlab` Python package provides array classes to represent arrays of MATLAB numeric types as Python variables so that MATLAB arrays can be passed between Python and MATLAB.

Create MATLAB Arrays in Python

You can create MATLAB numeric arrays in a Python session by calling constructors from the `matlab` Python package (for example, `matlab.double`, `matlab.int32`). The name of the constructor indicates the MATLAB numeric type.

You can use custom types for initializing MATLAB double arrays in Python. The custom type should inherit from the Python Abstract Base Class `collections.Sequence` to be used as an initializer.

You can pass MATLAB arrays as input arguments to functions called with the MATLAB Engine API for Python. When a MATLAB function returns a numeric array as an output argument, the engine returns the array to Python.

You can initialize the array with an optional `initializer` input argument that contains numbers. `initializer` must be a Python sequence type such as a `list`, `tuple`, or other sequence type. The optional `size` input argument sets the array size from a sequence. You can create multidimensional arrays by specifying `initializer` to contain multiple sequences of numbers, or by specifying `size` to be multidimensional. You can create a MATLAB array of complex numbers by setting the optional `is_complex` input argument to `True`. The `matlab` package provides the MATLAB array constructors listed in the table.

matlab Class	Constructor Call in Python
<code>matlab.double</code>	<code>matlab.double(initializer=None, size=None, is_complex=False)</code>
<code>matlab.single</code>	<code>matlab.single(initializer=None, size=None, is_complex=False)</code>
<code>matlab.int8</code>	<code>matlab.int8(initializer=None, size=None, is_complex=False)</code>
<code>matlab.int16</code>	<code>matlab.int16(initializer=None, size=None, is_complex=False)</code>
<code>matlab.int32</code>	<code>matlab.int32(initializer=None, size=None, is_complex=False)</code>
<code>matlab.int64^a</code>	<code>matlab.int64(initializer=None, size=None, is_complex=False)</code>
<code>matlab.uint8</code>	<code>matlab.uint8(initializer=None, size=None, is_complex=False)</code>
<code>matlab.uint16</code>	<code>matlab.uint16(initializer=None, size=None, is_complex=False)</code>
<code>matlab.uint32</code>	<code>matlab.uint32(initializer=None, size=None, is_complex=False)</code>
<code>matlab.uint64^b</code>	<code>matlab.uint64(initializer=None, size=None, is_complex=False)</code>
<code>matlab.logical</code>	<code>matlab.logical(initializer=None, size=None)^c</code>
<code>matlab.object</code>	No constructor. When a function returns a handle or a value object to a MATLAB object, the engine returns a <code>matlab.object</code> to Python.

- In Python 2.7 on Windows, `matlab.int64` is converted to `int32` in MATLAB. Also, MATLAB cannot return an `int64` array to Python.
- In Python 2.7 on Windows, `matlab.uint64` is converted to `uint32` in MATLAB. Also, MATLAB cannot return a `uint64` array to Python.
- Logicals cannot be made into an array of complex numbers.

When you create an array with N elements, the size is 1-by-N because it is a MATLAB array.

```
import matlab.engine
A = matlab.int8([1,2,3,4,5])
print(A.size)
```

```
(1, 5)
```

The initializer is a Python list containing five numbers. The MATLAB array size is 1-by-5, indicated by the tuple, (1,5).

MATLAB Array Attributes and Methods in Python

All MATLAB arrays created with `matlab` package constructors have the attributes and methods listed in this table.

Attribute or Method	Purpose
<code>size</code>	Size of array returned as a tuple
<code>reshape(size)</code>	Reshape array as specified by sequence size

Multidimensional MATLAB Arrays in Python

In Python, you can create multidimensional MATLAB arrays of any numeric type. Use two Python list variables to create a 2-by-5 MATLAB array of doubles.

```
import matlab.engine
A = matlab.double([[1,2,3,4,5], [6,7,8,9,10]])
print(A)
```

```
[[1.0,2.0,3.0,4.0,5.0],[6.0,7.0,8.0,9.0,10.0]]
```

The `size` attribute of `A` shows that it is a 2-by-5 array.

```
print(A.size)
```

```
(2, 5)
```

Index Into MATLAB Arrays in Python

You can index into MATLAB arrays just as you can index into Python list and tuple variables.

```
import matlab.engine
A = matlab.int8([1,2,3,4,5])
print(A[0])
```

```
[1,2,3,4,5]
```

The size of the MATLAB array is (1,5); therefore, A[0] is [1,2,3,4,5]. Index into the array to get 3.

```
print(A[0][2])
```

```
3
```

Python indexing is zero-based. When you access elements of MATLAB arrays in a Python session, use zero-based indexing.

Index into a multidimensional MATLAB array.

```
A = matlab.double([[1,2,3,4,5], [6,7,8,9,10]])
print(A[1][2])
```

```
8.0
```

Slice MATLAB Arrays in Python

You can slice MATLAB arrays the same way you slice Python list and tuple variables.

```
import matlab.engine
A = matlab.int8([1,2,3,4,5])
print(A[0][1:4])
```

```
[2,3,4]
```

You can assign data to a slice. This code shows assignment from a Python list to a slice of an array.

```
A = matlab.double([[1,2,3,4], [5,6,7,8]]);
A[0] = [10,20,30,40]
print(A)
```

```
[[10.0,20.0,30.0,40.0], [5.0,6.0,7.0,8.0]]
```

You can assign data from another MATLAB array, or from any Python iterable that contains numbers.

You can specify slices for assignment as shown here.

```
A = matlab.int8([1,2,3,4,5,6,7,8]);
A[0][2:4] = [30,40]
A[0][6:8] = [70,80]
print(A)

[[1,2,30,40,5,6,70,80]]
```

Note Slicing MATLAB arrays behaves differently from slicing a Python `list`. Slicing a MATLAB array returns a view instead of a shallow copy.

Given a MATLAB array and a Python `list` with the same values, assigning a slice results in different results as shown by the following code.

```
A = matlab.int32([[1,2],[3,4],[5,6]])
L = [[1,2],[3,4],[5,6]]
A[0] = A[0][::-1]
L[0] = L[0][::-1]
print(A)

[[2,2],[3,4],[5,6]]

print(L)

[[2, 1], [3, 4], [5, 6]]
```

Reshape MATLAB Arrays in Python

You can reshape a MATLAB array in Python with the `reshape` method. Input argument `size` must be a sequence that preserves the number of elements. Use `reshape` to change a 1-by-9 MATLAB array to 3-by-3.

```
import matlab.engine
A = matlab.int8([1,2,3,4,5,6,7,8,9])
A.reshape((3,3))
print(A)

[[1,4,7],[2,5,8],[3,6,9]]
```


See Also

Related Examples

- [“Use MATLAB Arrays in Python” on page 13-34](#)

Use MATLAB Arrays in Python

This example shows how to create a MATLAB array in Python and pass it as the input argument to the MATLAB `sqrt` function.

The `matlab` package provides constructors to create MATLAB arrays in Python. The MATLAB Engine API for Python can pass such arrays as input arguments to MATLAB functions, and can return such arrays as output arguments to Python. You can create arrays of any MATLAB numeric or logical type from Python sequence types.

Create a MATLAB array from a Python `list`. Call the `sqrt` function on the array.

```
import matlab.engine
eng = matlab.engine.start_matlab()
a = matlab.double([1,4,9,16,25])
b = eng.sqrt(a)
print(b)
```

```
[[1.0,2.0,3.0,4.0,5.0]]
```

The engine returns `b`, which is a 1-by-5 `matlab.double` array.

Create a multidimensional array. The `magic` function returns a 2-D `matlab.double` array to Python. Use a `for` loop to print each row on a separate line. (Press **Enter** again when you see the `...` prompt to close the loop and print.)

```
a = eng.magic(6)
for x in a: print(x)
...
[35.0,1.0,6.0,26.0,19.0,24.0]
[3.0,32.0,7.0,21.0,23.0,25.0]
[31.0,9.0,2.0,22.0,27.0,20.0]
[8.0,28.0,33.0,17.0,10.0,15.0]
[30.0,5.0,34.0,12.0,14.0,16.0]
[4.0,36.0,29.0,13.0,18.0,11.0]
```

Call the `tril` function to get the lower triangular portion of `a`. Print each row on a separate line.

```
b = eng.tril(a)
for x in b: print(x)
...
```

```
[35.0,0.0,0.0,0.0,0.0,0.0]
[3.0,32.0,0.0,0.0,0.0,0.0]
[31.0,9.0,2.0,0.0,0.0,0.0]
[8.0,28.0,33.0,17.0,0.0,0.0]
[30.0,5.0,34.0,12.0,14.0,0.0]
[4.0,36.0,29.0,13.0,18.0,11.0]
```

See Also

Related Examples

- “Call MATLAB Functions from Python” on page 13-14

More About

- “MATLAB Arrays as Python Variables” on page 13-28

Sort and Plot MATLAB Data from Python

This example shows how to sort data about patients into lists of smokers and nonsmokers in Python and plot blood pressure readings for the patients with MATLAB.

Start the engine, and read data about a set of patients into a MATLAB table. MATLAB provides a sample comma-delimited file, `patients.dat`, which contains information on 100 different patients.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.eval("T = readtable('patients.dat');",nargout=0)
```

The MATLAB `readtable` function reads the data into a table. The engine does not support the MATLAB table data type. However, with the MATLAB `table2struct` function you can convert the table to a scalar structure, which is a data type the engine does support.

```
eng.eval("S = table2struct(T,'ToScalar',true);",nargout=0)
eng.eval("disp(S)",nargout=0)
```

```
        LastName: {100x1 cell}
           Gender: {100x1 cell}
             Age: [100x1 double]
        Location: {100x1 cell}
           Height: [100x1 double]
           Weight: [100x1 double]
            Smoker: [100x1 double]
        Systolic: [100x1 double]
        Diastolic: [100x1 double]
SelfAssessedHealthStatus: {100x1 cell}
```

You can pass `S` from the MATLAB workspace into your Python session. The engine converts `S` to a Python dictionary, `D`.

```
D = eng.workspace["S"]
```

`S` has fields that contain arrays. The engine converts cell arrays to Python `list` variables, and numeric arrays to MATLAB arrays. Therefore, `D["LastName"]` is of data type `list`, and `D["Age"]` is of data type `matlab.double`.

Sort blood pressure readings into lists of smokers and nonsmokers. In `patients.dat`, the column `Smoker` indicated a smoker with logical 1 (true), and a nonsmoker with a logical 0 (false). Convert `D["Smoker"]` to a `matlab.logical` array for sorting.

```
smoker = matlab.logical(D["Smoker"])
```

Convert the Diastolic blood pressure readings and Smoker indicators into 1-by-100 MATLAB arrays for sorting.

```
pressure = D["Diastolic"]
pressure.reshape((1,100))
pressure = pressure[0]
smoker.reshape((1,100))
smoker = smoker[0]
```

Sort the pressure array into lists of blood pressure readings for smokers and non-smokers. Python list comprehensions provide a compact method for iterating over sequences. With the Python zip function, you can iterate over multiple sequences in a single for loop.

```
sp = [p for (p,s) in zip(pressure,smoker) if s is True]
nsp = [p for (p,s) in zip(pressure,smoker) if s is False]
```

Display the length of sp, the blood pressure readings for smokers in a list.

```
print(len(sp))
```

```
34
```

Display the length of nsp, the list of readings for nonsmokers.

```
print(len(nsp))
```

```
66
```

Calculate the mean blood pressure readings for smokers and nonsmokers. Convert sp and nsp to MATLAB arrays before passing them to the MATLAB mean function.

```
sp = matlab.double(sp)
nsp = matlab.double(nsp)
print(eng.mean(sp))
```

```
89.9117647059
```

Display the mean blood pressure for the nonsmokers.

```
print(eng.mean(nsp))
```

```
79.3787878788
```

Plot blood pressure readings for the smokers and nonsmokers. To define two x-axes for plotting, call the MATLAB `linspace` function. You can plot the 34 smokers and 66 nonsmokers on the same scatter plot.

```
sdx = eng.linspace(1.0,34.0,34)
nsdx = eng.linspace(1.0,34.0,66)
```

Show the axes boundaries with the `box` function.

```
eng.figure(nargout=0)
eng.hold("on",nargout=0)
eng.box("on",nargout=0)
```

You must call the `figure`, `hold`, and `box` functions with `nargout=0`, because these functions do not return output arguments.

Plot the blood pressure readings for the smokers and nonsmokers, and label the plot. For many MATLAB functions, the engine can return a handle to a MATLAB graphics object. You can store a handle to a MATLAB object in a Python variable, but you cannot manipulate the object properties in Python. You can pass MATLAB objects as input arguments to other MATLAB functions.

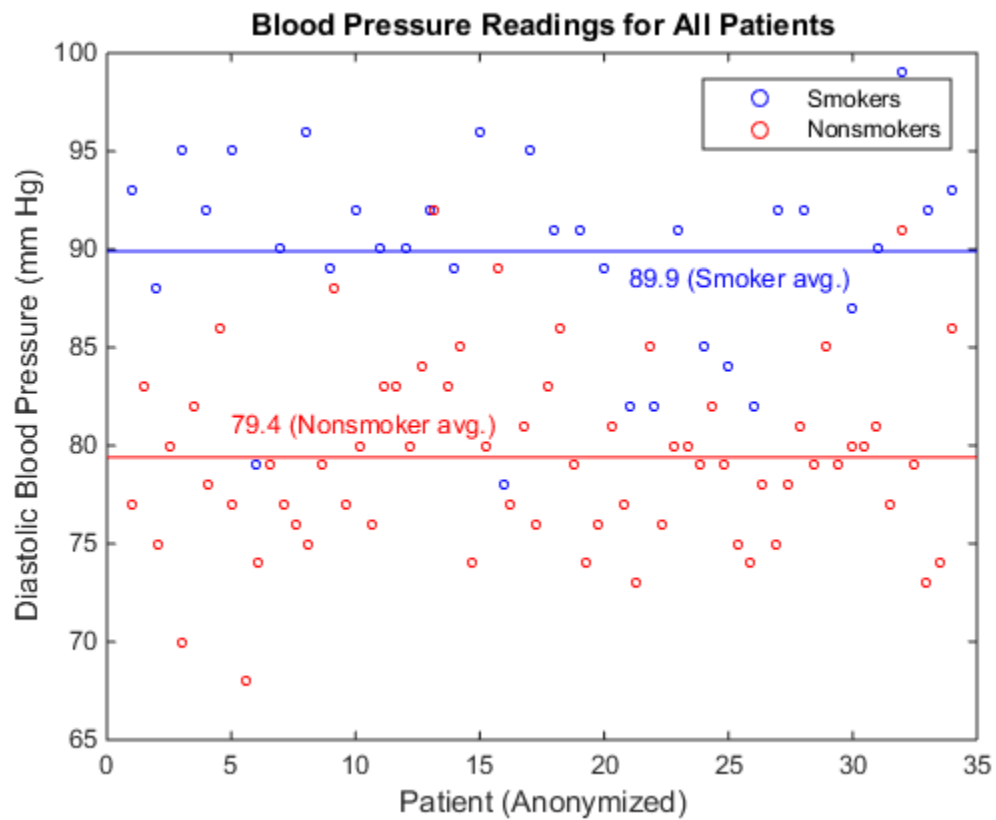
```
eng.scatter(sdx,sp,10,'blue')
<matlab.object object at 0x22d1510>
```

In the rest of this example, assign the output argument of MATLAB functions to `h` as a placeholder.

```
h = eng.scatter(nsdx,nsp,10,'red')
h = eng.xlabel("Patient (Anonymized)")
h = eng.ylabel("Diastolic Blood Pressure (mm Hg)")
h = eng.title("Blood Pressure Readings for All Patients")
h = eng.legend("Smokers","Nonsmokers")
```

Draw lines showing the average blood pressure readings for smokers and nonsmokers.

```
x = matlab.double([0,35])
y = matlab.double([89.9,89.9])
h = eng.line(x,y,"Color","blue")
h = eng.text(21.0,88.5,"89.9 (Smoker avg.)","Color","blue")
y = matlab.double([79.4,79.4])
h = eng.line(x,y,"Color","red")
h = eng.text(5.0,81.0,"79.4 (Nonsmoker avg.)","Color","red")
```



See Also

`readtable | scatter`

Get Help for MATLAB Functions from Python

In this section...
“How to Find MATLAB Help” on page 13-40
“Open MATLAB Help Browser from Python” on page 13-40
“Display MATLAB Help at Python Prompt” on page 13-41

How to Find MATLAB Help

From Python, you can access supporting documentation for all MATLAB functions. This documentation includes examples and describes input arguments, output arguments, and calling syntax for each function.

The MATLAB Engine API for Python enables you to use the MATLAB `doc` and `help` functions. Use `doc` to open the MATLAB Help browser. Use `help` to get a brief description of a MATLAB function at the Python prompt.

Open MATLAB Help Browser from Python

From Python, you can use the Help browser to open MATLAB function reference pages and search the documentation.

For example, display the reference page for the MATLAB `plot` function. (Since `doc` returns no output arguments, you must set `nargout=0`.)

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.doc("plot",nargout=0)
```

The reference page includes a description of the function, examples, and links to related documentation.

Note Click an example title, or on the arrow next to a title, if you do not see the examples on a MATLAB reference page. Examples can be collapsed or expanded within a page.

If you call `eng.doc` with no positional arguments, it opens the Help browser. (You still must set the keyword argument `nargout=0`).


```
eng.doc(nargout=0)
```

To search the MATLAB documentation, type an expression in the search box at the top of any page in the Help browser. The browser returns a list of search results, highlighting words that match the expression.

Alternatively, you can search the documentation with the `docsearch` function. For example, search for pages that mention `plot`.

```
eng.docsearch("plot",nargout=0)
```

Display MATLAB Help at Python Prompt

To display help text for a function at the Python prompt, call the MATLAB `help` function. For example, display the help text for `erf`.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.help("erf",nargout=0)
```

```
ERF Error function.
```

```
Y = ERF(X) is the error function for each element of X. X must be
real. The error function is defined as:
```

```
erf(x) = 2/sqrt(pi) * integral from 0 to x of exp(-t^2) dt.
```

```
See also ERFC, ERFCX, ERFINV, ERFCINV.
```

```
Other functions named erf:
```

```
codistributed/erf
gpuArray/erf
sym/erf
```

```
Reference page in Help browser
doc erf
```

The output displays the help text, but does not include any links to help for other MATLAB functions that might be mentioned.

Default Numeric Types in MATLAB and Python

MATLAB stores all numeric values as double-precision floating point numbers by default. In contrast, Python stores some numbers as integers by default. Because of this difference, you might pass integers as input arguments to MATLAB functions that expect double-precision numbers.

Consider these variable assignments in MATLAB:

```
x = 4;  
y = 4.0;
```

Both `x` and `y` are of data type `double`. Now consider the same assignments in Python:

```
x = 4  
y = 4.0
```

`x` and `y` are of different numeric data types.

```
print(type(x))
```

```
<type 'int'>
```

```
print(type(y))
```

```
<type 'float'>
```

Most MATLAB functions take numeric input arguments of data type `double`. The best practice is to ensure that numbers you pass as input arguments to MATLAB functions are of Python data type `float`, not Python data type `int`. You can ensure that Python variables are floating point numbers if you:

- Make literals floating point numbers. For example, type `4.0` instead of `4`.
- Convert to data type `float`. For example, `x = float(4)` casts the number to data type `float`.
- Create a `matlab.double` array from a number or sequence. For example, `x = matlab.double([1,2,3,4,5])` creates an array of MATLAB data type `double` from a list of Python integers.

When you pass an integer to a MATLAB function that takes an input argument of data type `double`, the engine raises an error. See “MatlabExecutionError: Undefined Function” on page 13-47 for an example.

When you call a MATLAB function that does take integers as numeric input arguments, you can pass input arguments of Python data type `int` to the function.

System Requirements for MATLAB Engine API for Python

In this section...
“Python Version Support” on page 13-44
“Download 64-Bit Versions of Python and MATLAB” on page 13-44
“Requirements for Building Python from Source” on page 13-45

You can use the MATLAB Engine API for Python on any platform that MATLAB supports.

Python Version Support

To use the MATLAB Engine API for Python, you must have a supported version of the reference Python implementation (also known as CPython) installed on your system. MATLAB supports versions 2.7, 3.6, and 3.7.

To download and install Python, see “Install Supported Python Implementation” on page 20-3.

Note For 64-bit MATLAB on Microsoft Windows systems, select the 64-bit Python version, called Windows x86-64 MSI installer.

To call Python from your operating system prompt, do one of the following.

- Add the full path to Python to your PATH environment variable
- Include the full path when you call the Python interpreter

To determine if you are calling a supported version, type `python -V` at your operating system prompt to display the Python version number.

For help on the Python language, see www.python.org/doc in the python.org documentation. For help on third-party or user-defined modules, refer to the product documentation.

Download 64-Bit Versions of Python and MATLAB

The architecture of MATLAB must match the architecture of Python. On the Python download site, downloads for Microsoft Windows platforms are 32-bit versions by default.

To download the 64-bit version, choose options with the name *Windows x86-64 MSI installer*.

To test whether your version of Python is 32-bit or 64-bit, type the following code at the Python prompt:

```
import sys
print(sys.maxsize > 2**32)
```

This code returns `True` if the Python interpreter is 64-bit, and `False` if it is 32-bit. (For more details, see [Python 2.7 Documentation — Cross Platform](#).)

Requirements for Building Python from Source

To enable wide-unicode support for Python 2.7 on Linux, configure the build with the `--enable-unicode=ucs4` option. This configure option is not needed when you build any version of Python on Mac systems, or Python 3.x on Linux.

See Also

More About

- “Install Supported Python Implementation” on page 20-3

External Websites

- www.python.org/doc

Limitations to MATLAB Engine API for Python

Features Not Supported in MATLAB

- The engine cannot start or connect to MATLAB on a remote machine.
- Python keyword arguments cannot be input arguments to MATLAB functions called with the engine. The engine passes only positional arguments to MATLAB functions.
- The size of data arrays passed between Python and MATLAB is limited to 2 GB. This limit applies to the data plus supporting information passed between the processes.
- A recursive data structure cannot be passed as an input argument to a MATLAB function, or put into an engine workspace. (A recursive data structure is a Python data structure that includes itself as a value.)

See Also

More About

- “Troubleshoot MATLAB Errors in Python” on page 13-47

Troubleshoot MATLAB Errors in Python

In this section...

“MATLAB Errors in Python” on page 13-47

“MatlabExecutionError: Undefined Function” on page 13-47

“SyntaxError: Expression Not Valid Target” on page 13-48

“SyntaxError: Invalid Syntax” on page 13-48

“Cannot Find MATLAB Session in Python” on page 13-49

MATLAB Errors in Python

When a MATLAB function raises an error, the MATLAB Engine for Python stops the function and catches the exception raised by MATLAB. The engine copies the error message to a new Python exception. The engine raises the Python exception.

If the Python interpreter catches the exception, the interpreter displays the error message that came from MATLAB. You also can handle exceptions raised by the engine in your Python code. See the `matlab.engine.MatlabEngine` and `matlab.engine.FutureResult` reference pages for the types of exceptions that the engine can raise.

MatlabExecutionError: Undefined Function

Call the MATLAB `sqrt` function on an integer from Python. (This code sample omits the Python trace back and shows the error message only.)

```
import matlab.engine
eng = matlab.engine.start_matlab()
print(eng.sqrt(4))
```

```
matlab.engine.MatlabExecutionError: Undefined function 'sqrt' for input arguments of type 'int64'.
```

MATLAB defines a `sqrt` function, but expects the input argument to be of data type `double`, not an integer. However, the input argument is `4`, and before it is passed to MATLAB, Python interprets `4` as an integer. The engine converts the Python integer to an `int64` MATLAB data type.

MATLAB and Python define different default types for numbers. If you type `x = 4` at the MATLAB command line, `x` is a MATLAB `double`. If you type `x = 4` at the Python command line, `x` is a Python `int`.

To avoid this error, specify input arguments that are of Python data type `float`. The engine converts this type to MATLAB `double`.

```
print(eng.sqrt(4.0))
```

```
2.0
```

SyntaxError: Expression Not Valid Target

You can call the MATLAB `eval` function from Python to create MATLAB variables. (This code sample omits the Python trace back and shows the error message only.)

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.eval("x = 4;")
```

```
SyntaxError: Error: The expression to the left of the equals sign is not a
valid target for an assignment.
```

When the engine calls `eval`, it passes a statement to MATLAB for execution. When you do not specify the input argument `nargout` input argument, the engine expects one output argument. However, this MATLAB statement returns no output arguments.

To avoid this error, specify `nargout` as 0 whenever the MATLAB function you call returns no output arguments.

```
eng.eval("x = 4;", nargout=0)
```

SyntaxError: Invalid Syntax

Call the MATLAB `print` function from Python 2.7 to print a plot you create with the MATLAB `surf` function.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.eval("surf(peaks)", nargout=0)
eng.print("-djpeg", "surf", nargout=0)
```

```
File "<stdin>", line 1
eng.print("-djpeg", "surf", nargout=0)
```



```

^
SyntaxError: invalid syntax

```

If MATLAB and Python functions have the same name, then the engine calls the MATLAB function.

However, the engine cannot directly call a MATLAB function that has a name that also is a reserved word in the Python language. For example, in Python 2.7, `print` is a reserved word. (In Python 3.x, the previous code runs because `print` is a built-in function, not a reserved word.)

To avoid this error, call the MATLAB function with `eval`.

```
eng.eval("print('-djpeg','surf');",nargout=0)
```

If the MATLAB function is a function that you created, you can rename it so that its name is no longer a Python reserved word. The Python documentation lists reserved words:

- Python 2.7 reserved words (https://docs.python.org/2/reference/lexical_analysis.html#keywords)
- Python 3.x reserved words (https://docs.python.org/3/reference/lexical_analysis.html#keywords)

Cannot Find MATLAB Session in Python

If you override the operating system `TEMP` or `TMP` environment variables in MATLAB, Python might not be able to connect to the MATLAB Engine for Python. For example, if you type the following at the Python prompt:

```
matlab.engine.find_matlab()
```

Python displays `()`.

MATLAB Engine for Python uses the `temp` folder to record information for shared MATLAB sessions. To work around this issue, make the following changes to the environment variables in Python. *temp_folder* is the path to the folder which you set in MATLAB.

```
os.environ['TMP'] = r'temp_folder'
os.environ['TEMP'] = r'temp_folder'
eng=matlab.engine.find_matlab()
```

See Also

More About

- “Limitations to MATLAB Engine API for Python” on page 13-46
- “Default Numeric Types in MATLAB and Python” on page 13-42

Engine API for C++

- “Introduction to Engine API for C++” on page 14-2
- “C++ Engine API” on page 14-5
- “Build C++ Engine Programs” on page 14-9
- “Test Your Build Environment” on page 14-12
- “Start MATLAB Sessions from C++” on page 14-14
- “Connect C++ to Running MATLAB Session” on page 14-16
- “Call MATLAB Functions from C++” on page 14-19
- “Evaluate MATLAB Statements from C++” on page 14-29
- “Pass Variables from C++ to MATLAB” on page 14-32
- “Pass Variables from MATLAB to C++” on page 14-34
- “Redirect MATLAB Command Window Output to C++” on page 14-38
- “Create Cell Arrays from C++” on page 14-41
- “Create Structure Arrays from C++” on page 14-46
- “Pass Enumerations to MATLAB From C++” on page 14-50
- “Pass Sparse Arrays to MATLAB From C++” on page 14-53
- “Run Simulink Simulation from C++” on page 14-55
- “Convert C++ Engine Application to MATLAB Compiler SDK Application” on page 14-59

Introduction to Engine API for C++

In this section...
“Getting Started” on page 14-2
“Basic Elements of C++ Engine Programs” on page 14-2

The MATLAB Engine API for C++ enables C++ programs to interact with MATLAB synchronously or asynchronously. Supported operations include:

- Start MATLAB.
- Connect to a MATLAB shared session on the local machine.
- Call MATLAB functions with input arguments passed from C++ and output variables returned from MATLAB.
- Evaluate MATLAB statements in the MATLAB base workspace.
- Pass variables from C++ to MATLAB and from MATLAB to C++.

The MATLAB Engine API for C++ is included in the MATLAB product. For the complete API, see “C++ Engine API” on page 14-5.

Getting Started

The MATLAB Engine API for C++ comprises a set of C++ header files and C runtime shared libraries. The namespace `matlab::engine` contains several utility functions and a set of C++ classes.

Begin using the MATLAB Engine API for C++ by setting up your build and runtime environment. Ensure that you have a supported compiler installed. Use the MATLAB `mex` command to setup your environment and to build C++ applications. You can also configure your IDE to build C++ applications that use the Engine API. For information on how to do this, see “Build C++ Engine Programs” on page 14-9.

The Engine API supports the use of the MATLAB Data API. This API provides a way for applications running outside of MATLAB to work with MATLAB data. For more information on this API, see “MATLAB Data API”.

Basic Elements of C++ Engine Programs

Here is some simple C++ engine code showing the basic elements used to execute a MATLAB command. This code passes a vector of data arrays to a MATLAB function,

`movsum`, and returns the result. This C++ code executes the equivalent of these statements in MATLAB.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movsum(A,3,'Endpoints','discard');
```

Basic Elements of C++ Engine Code

Add header files for MATLAB engine and MATLAB data arrays.

```
#include "MatlabEngine.hpp"
#include "MatlabDataArray.hpp"
```

Start a MATLAB session and get a unique pointer to the instance.

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

Create a MATLAB data array factory to construct the data types used by the `matlab::engine::MATLABEngine` member functions.

```
matlab::data::ArrayFactory factory;
```

Define a vector of MATLAB data arrays for the input arguments to the MATLAB function. Each argument is an array in the vector.

```
// Create a vector of MATLAB data arrays for arguments
std::vector<matlab::data::Array> args({
    factory.createArray<double>({ 1, 10 }, { 4, 8, 6, -1, -2, -3, -1, 3, 4, 5 }),
    factory.createScalar<int32_t>(3),
    factory.createCharArray("Endpoints"),
    factory.createCharArray("discard")
});
```

Call the MATLAB `movsum` function using the `MATLABEngine::feval` member function. Define the returned result as a MATLAB data array of the appropriate type.

```
// Call MATLAB function with arguments and return results
matlab::data::TypedArray<double> result = matlabPtr->feval(u"movsum", args);
```

See Also

`matlab::data::ArrayFactory` | `matlab::engine::MATLABEngine`

Related Examples

- “Test Your Build Environment” on page 14-12

- “Build C++ Engine Programs” on page 14-9
- “Start MATLAB Sessions from C++” on page 14-14
- “Call MATLAB Functions from C++” on page 14-19

C++ Engine API

In this section...
“Utility Functions” on page 14-5
“Classes” on page 14-6
“MATLABEngine Member Functions” on page 14-6
“Exception Classes” on page 14-7
“Data Size Limitations” on page 14-8
“Using Engine in Multiple-Thread Environment” on page 14-8

The MATLAB Engine API for C++ comprises functions, classes, and typedefs in `matlab::engine` namespace. This API supports the MATLAB Data API, which provides a way for applications running outside of MATLAB to work with MATLAB data through a MATLAB-neutral interface. For more information on the MATLAB Data API, see “MATLAB Data API”.

Utility Functions

Function	Purpose
<code>matlab::engine::startMATLAB</code>	Start a MATLAB session
<code>matlab::engine::startMATLABAsync</code>	Start a MATLAB session asynchronously
<code>matlab::engine::connectMATLAB</code>	Connect to a shared MATLAB session on the local machine using the specified name.
<code>matlab::engine::connectMATLABAsync</code>	Connect to a shared MATLAB session on the local machine using the specified name asynchronously.
<code>matlab::engine::findMATLAB</code>	Find all shared MATLAB sessions from the local machine.
<code>matlab::engine::findMATLABAsync</code>	Find all shared MATLAB sessions from the local machine asynchronously.
<code>matlab::engine::convertUTF8StringToUTF16String</code>	Convert UTF-8 string to UTF-16 string.

Function	Purpose
<code>matlab::engine::convertUTF16StringToUTF8String</code>	Convert UTF-16 string to UTF-8 string.
<code>matlab::engine::terminateEngineClient</code>	Free engine resources during runtime

Classes

Class	Purpose
<code>matlab::engine::MATLABEngine</code>	Use to execute MATLAB functions from C++.
<code>matlab::engine::FutureResult</code>	Retrieve results from asynchronous operations.
<code>matlab::engine::WorkspaceType</code>	Enumeration class defining MATLAB workspace as BASE or GLOBAL

MATLABEngine Member Functions

The `matlab::engine::MATLABEngine` class defines the following member functions.

Member Function	Purpose
<code>"feval"</code>	Evaluate a MATLAB® function with arguments synchronously.
<code>"fevalAsync"</code>	Evaluate a MATLAB® function with arguments asynchronously.
<code>"eval"</code>	Evaluate a MATLAB® statement synchronously.
<code>"evalAsync"</code>	Evaluate a MATLAB® statement asynchronously.
<code>"getVariable"</code>	Get a variable from the MATLAB® base or global workspace synchronously.
<code>"getVariableAsync"</code>	Get a variable from the MATLAB® base or global workspace asynchronously.

Member Function	Purpose
"setVariable"	Put a variable into the MATLAB® base or global workspace synchronously.
"setVariableAsync"	Put a variable into the MATLAB® base or global workspace asynchronously.
"getProperty"	Get an object property value.
"getPropertyAsync"	Get an object property value asynchronously.
"setProperty"	Set an object property value.
"setPropertyAsync"	Set an object property value asynchronously.

Exception Classes

Exception	Cause
<code>matlab::engine::Exception</code>	Base class of all C++ Engine exceptions.
<code>matlab::engine::EngineException</code>	There is a MATLAB runtime error in function or MATLAB fails to start.
<code>matlab::engine::MATLABNotAvailableException</code>	The MATLAB session is not available
<code>matlab::engine::MATLABSyntaxException</code>	There is a syntax error in the MATLAB function.
<code>matlab::engine::MATLABExecutionException</code>	There is a MATLAB runtime error in the MATLAB function or statement.
<code>matlab::engine::CanceledException</code>	Evaluation of the MATLAB function is canceled.
<code>matlab::engine::InterruptedException</code>	Thrown by <code>matlab::engine::FutureResult::get</code> if the evaluation of the MATLAB function or statement is interrupted.
<code>matlab::engine::TypeConversionException</code>	The result of the MATLAB function cannot be converted to the specified type

Data Size Limitations

The size of data arrays passed between C++ and MATLAB is limited to 2 GB. This limit applies to the data plus supporting information passed between the processes.

Using Engine in Multiple-Thread Environment

The MATLAB Engine for C++ is safe to use in a multiple-thread environment. You can make these connections to shared MATLAB sessions:

- Connect to different shared MATLAB sessions from separate threads of a C++ application.
- Connect to a single MATLAB session from multiple engine applications.

You cannot use multiple threads of the same process to connect to a single shared MATLAB session.

See Also

Related Examples

- “Build C++ Engine Programs” on page 14-9

Build C++ Engine Programs

In this section...

“Supported Compilers” on page 14-9

“Build .cpp File with mex Command” on page 14-9

“General Requirements” on page 14-9

“Engine Include Files” on page 14-10

“Runtime Environment” on page 14-11

Supported Compilers

Use compilers that support C++11. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.

Build .cpp File with mex Command

If you have installed one of the supported compilers, set up the compiler for C++ engine applications using the `mex` command. When provided with an option to select a compiler, select an installed compiler that the MATLAB Engine API for C++ supports.

```
mex -setup -client engine C++
```

Build your C++ engine program using the MATLAB `mex` command.

```
mex -client engine MyEngineCode.cpp
```

To test your setup, see “Test Your Build Environment” on page 14-12 .

General Requirements

Set up your environment for building and running C++ engine applications using these libraries, include files, environment variables. Engine applications require the engine library `libMatlabEngine`, the MATLAB Data Array library `libMatlabDataArray`, and supporting include files.

In the following sections, replace `matlabroot` with the path returned by the MATLAB `matlabroot` command.

Windows Libraries

In these path specifications, replace *compiler* with either *microsoft* or *mingw64*.

- Engine library — *matlabroot\extern\lib\win64\compiler\libMatlabEngine.lib*
- MATLAB Data Array library — *matlabroot\extern\lib\win64\compiler\libMatlabDataArray.lib*

Linux Libraries

- Engine library — *matlabroot/extern/bin/glnxa64/libMatlabEngine.so*
- MATLAB Data Array library — *matlabroot/extern/bin/glnxa64/libMatlabDataArray.so*

Additional library — *pthread*

For example, to build *myEngineApp.cpp*, use these libraries. Replace *matlabroot* with the path returned by the MATLAB *matlabroot* command.

```
g++ -std=c++11 -I <matlabroot>/extern/include/ -L <matlabroot>/extern/bin/glnxa64/  
-pthread myEngineApp.cpp -lMatlabDataArray -lMatlabEngine
```

Mac Libraries

- Engine library — *matlabroot/extern/bin/maci64/libMatlabEngine.dylib*
- MATLAB Data Array library — *matlabroot/extern/bin/maci64/libMatlabDataArray.dylib*

Engine Include Files

Header files contain function declarations with prototypes for the routines that you access in the API libraries. These files are in the *matlabroot/extern/include* folder and are the same for Windows, Mac, and Linux systems. Engine applications use:

- *MatlabEngine.hpp* — Definitions for the C++ engine API
- *MatlabDataArray.hpp* — Definitions for MATLAB Data Arrays

MATLAB Data Array is a collection of classes and APIs that provide a generic interface between external data and MATLAB.

Runtime Environment

This table lists the names of the environment variables and the paths to add for the respective platforms.

Operating System	Variable	Path
Windows	PATH	<i>matlabroot</i> \extern\bin\win64
64-bit Apple Mac	DYLD_LIBRARY_PATH	<i>matlabroot</i> /extern/bin/maci64
64-bit Linux	LD_LIBRARY_PATH	<i>matlabroot</i> /extern/bin/ glnxa64: <i>matlabroot</i> /sys/os/ glnxa64

See Also

`matlab::engine::MATLABEngine | mex`

Related Examples

- “C++ Engine API” on page 14-5
- “Test Your Build Environment” on page 14-12

Test Your Build Environment

To test your installation and environment, save the following C++ code in a file named `testFeval.cpp` (you can use any name). To build the engine application, use these commands from your command window:

```
mex -setup -client engine C++
```

Select the installed compiler you want to use when prompted by the `mex setup` script. Then call the `mex` command to build your program. Ensure that the MATLAB Engine API for C++ supports the compiler you select. For an up-to-date list of supported compilers, see the [Supported and Compatible Compilers](#) website.

```
mex -v -client engine testFeval.cpp
```

The `mex` command saves the executable file in the same folder.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>
void callSQRT() {

    using namespace matlab::engine;

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

    //Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Define a four-element typed array
    matlab::data::TypedArray<double> const argArray =
        factory.createArray({ 1,4 }, { -2.0, 2.0, 6.0, 8.0 });

    // Call MATLAB sqrt function on the data array
    matlab::data::Array const results = matlabPtr->feval(u"sqrt", argArray);

    // Display results
    for (int i = 0; i < results.getNumberOfElements(); i++) {
        double a = argArray[i];
        std::complex<double> v = results[i];
        double realPart = v.real();
        double imgPart = v.imag();
        std::cout << "Square root of " << a << " is " <<
            realPart << " + " << imgPart << "i" << std::endl;
    }
}

int main() {
    callSQRT();
    return 0;
}
```

Here is the output from this program. In this case, MATLAB returns a complex array because one of the numbers in the data array is negative.

```
Square root of -2 is 0 + 1.41421i  
Square root of 2 is 1.41421 + 0i  
Square root of 6 is 2.44949 + 0i  
Square root of 8 is 2.82843 + 0i
```

See Also

`matlab::engine::MATLABEngine` | `mex`

Related Examples

- “C++ Engine API” on page 14-5
- “Build C++ Engine Programs” on page 14-9
- “Call MATLAB Functions from C++” on page 14-19

Start MATLAB Sessions from C++

In this section...

“Start MATLAB Session Synchronously” on page 14-14

“Start MATLAB Session Asynchronously” on page 14-14

“Start MATLAB with Startup Options” on page 14-15

Start a MATLAB engine session from your C++ program synchronously or asynchronously. To start the session, use one of these utility functions, which are defined in the `matlab::engine` namespace:

- `matlab::engine::startMATLAB` — Start a MATLAB session synchronously.
- `matlab::engine::startMATLABAsync` — Start a MATLAB session asynchronously.

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Start MATLAB Session Synchronously

Start MATLAB from C++ synchronously. `startMATLAB` returns a unique pointer to the `MATLABEngine` instance.

```
#include "MatlabEngine.hpp"

void startMLSession() {
    using namespace matlab::engine;

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

Start MATLAB Session Asynchronously

Start MATLAB from C++ asynchronously. Use `FutureResult::get` to get the unique pointer to the `MATLABEngine` instance that is returned by `startMATLABAsync`.

```
#include "MatlabEngine.hpp"

void startMLSessionAsync() {
    using namespace matlab::engine;
```



```
// Start MATLAB engine asynchronously
FutureResult<std::unique_ptr<MATLABEngine>> matlabFuture = startMATLABAsync();
std::unique_ptr<MATLABEngine> matlabPtr = matlabFuture.get();
}
```

Start MATLAB with Startup Options

You can start a MATLAB session using supported MATLAB startup options. For information on MATLAB startup options, see “Commonly Used Startup Options”. For information on the startup options supported by the engine, see `matlab::engine::MATLABEngine`.

This sample code starts MATLAB using the `-r` and `matlab.engine.ShareEngine` options. Create a vector containing each option as an element in the vector.

```
#include "MatlabEngine.hpp"

void startMLOptions() {
    using namespace matlab::engine;

    // Start MATLAB with nojvm option
    std::vector<String> optionVec;
    optionVec.push_back(u"-r");
    optionVec.push_back(u"matlab.engine.shareEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB(optionVec);
}
```

See Also

`matlab::engine::startMATLAB` | `matlab::engine::startMATLABAsync`

Related Examples

- “Connect C++ to Running MATLAB Session” on page 14-16

Connect C++ to Running MATLAB Session

In this section...

- “Connect to Shared MATLAB” on page 14-16
- “Connect to Shared MATLAB Asynchronously” on page 14-17
- “Specify Name of Shared Session” on page 14-17
- “Find and Connect to Named Shared Session” on page 14-18

You can connect the C++ engine to shared MATLAB sessions that are running on the local machine. To connect to a shared MATLAB session:

- Start MATLAB as a shared session, or make a running MATLAB process shared using the `matlab.engine.shareEngine` MATLAB function.
- Find the names of the MATLAB shared sessions using `matlab::engine::findMATLAB` or `matlab::engine::findMATLABAsync`.
- Pass a `matlab::engine::String` containing the name of the shared MATLAB session to the `matlab::engine::connectMATLAB` or `matlab::engine::connectMATLABAsync` member function. These functions connect the C++ engine to the shared session.

If you do not specify the name of a shared MATLAB session when calling `matlab::engine::connectMATLAB` or `matlab::engine::connectMATLABAsync`, the engine uses the first shared MATLAB session created. If there are no shared MATLAB sessions available, the engine creates a shared MATLAB session and connects to this session.

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Connect to Shared MATLAB

This sample code connects to the first shared MATLAB session found.

```
#include "MatlabEngine.hpp"

void syncConnect() {
    using namespace matlab::engine;

    // Connect to shared MATLAB session
```

```

    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB();
}

```

Connect to Shared MATLAB Asynchronously

This sample code connects to the first shared MATLAB session found asynchronously.

```

#include "MatlabEngine.hpp"

void asyncConnect() {
    using namespace matlab::engine;

    // Find and connect to shared MATLAB session
    FutureResult<std::unique_ptr<MATLABEngine>> futureMATLAB = connectMATLABAsync();
    ...
    std::unique_ptr<MATLABEngine> matlabPtr = futureMATLAB.get();
}

```

Specify Name of Shared Session

You can specify the name of the shared MATLAB session when you execute the `matlab.engine.shareEngine` MATLAB function. Doing so eliminates the need to use `matlab::engine::findMATLAB` or `matlab::engine::findMATLABAsync` to find the name.

For example, start MATLAB and name the shared session `myMatlabEngine`.

```
matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
```

This sample code connects to the MATLAB session named `myMatlabEngine` from C++.

Note Start the named MATLAB session before connecting from the C++ code.

```

#include "MatlabEngine.hpp"

void connectToML() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(u"myMatlabEngine");
}

```

Find and Connect to Named Shared Session

To connect to a named MATLAB shared session, use `matlab::engine::findMATLAB` or `matlab::engine::findMATLABAsync` to find the names of all available named MATLAB shared sessions.

This sample code tries to find a MATLAB shared session named `myMatlabEngine` and connects to it if the session is found.

```
void findNConnect() {
    using namespace matlab::engine;

    // Find and connect to shared MATLAB session
    std::unique_ptr<MATLABEngine> matlabPtr;
    std::vector<String> names = findMATLAB();
    std::vector<String>::iterator it;
    it = std::find(names.begin(), names.end(), u"myMatlabEngine");
    if (it != names.end()) {
        matlabPtr = connectMATLAB(*it);
    }

    // Determine if engine connected
    if (matlabPtr){
        matlab::data::ArrayFactory factory;
        matlab::data::CharArray arg = factory.createCharArray("-release");
        matlab::data::CharArray version = matlabPtr->feval(u"version", arg);
        std::cout << "Connected to: " << version.toAscii() << std::endl;
    }
    else {
        std::cout << "myMatlabEngine not found" << std::endl;
    }
}
```

See Also

`matlab.engine.shareEngine` | `matlab::engine::connectMATLAB` |
`matlab::engine::connectMATLABAsync` | `matlab::engine::findMATLAB` |
`matlab::engine::findMATLABAsync`

Related Examples

- “Start MATLAB Sessions from C++” on page 14-14

Call MATLAB Functions from C++

In this section...

“Call Function with Single Returned Argument” on page 14-20

“Call Function with Name/Value Arguments” on page 14-22

“Call Function Asynchronously” on page 14-23

“Call Function with Multiple Returned Arguments” on page 14-23

“Call Function with Native C++ Types” on page 14-24

“Control Number of Outputs” on page 14-25

Call MATLAB functions from C++ using the “feval” and “fevalAsync” member functions of the `matlab::engine::MATLABEngine` class. Use these functions when you want to pass function arguments from C++ to MATLAB and to return the result of the function execution to C++. These member functions work like the MATLAB `feval` function.

To call a MATLAB function:

- Pass the function name as a `matlab::engine::String`.
- Define the input arguments required by the MATLAB function. You can use either native C++ data types or the MATLAB Data API. For more information, see “MATLAB Data API”.
- Specify the number of outputs expected from the MATLAB function. One output is the default. For more information, see “Call Function with Multiple Returned Arguments” on page 14-23 and “Control Number of Outputs” on page 14-25.
- Define the appropriate returned type for the results of the MATLAB function.
- Use stream buffers to redirect standard output and standard error from the MATLAB command window to C++. For more information, see “Redirect MATLAB Command Window Output to C++” on page 14-38

To evaluate MATLAB statements using variables in the MATLAB base workspace, use the `matlab::engine::MATLABEngine` “eval” and “evalAsync” member functions. These functions enable you to create and use variables in the MATLAB workspace, but do not return values. For more information, see “Evaluate MATLAB Statements from C++” on page 14-29.

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Call Function with Single Returned Argument

This example uses the MATLAB `gcd` function to find the greatest common divisor of two numbers. The `MATLABEngine::feval` member function returns the results of the `gcd` function call.

Use the `matlab::data::ArrayFactory` to create two scalar `int16_t` arguments. Pass the arguments to `MATLABEngine::feval` in a `std::vector`.

```
#include "MatlabEngine.hpp"
#include "MatlabDataArray.hpp"
#include <iostream>

void callFevalgcd() {
    // Pass vector containing MATLAB data array scalar
    using namespace matlab::engine;

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

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Pass vector containing 2 scalar args in vector
    std::vector<matlab::data::Array> args({
        factory.createScalar<int16_t>(30),
        factory.createScalar<int16_t>(56) });

    // Call MATLAB function and return result
    matlab::data::TypedArray<int16_t> result = matlabPtr->feval(u"gcd", args);
    int16_t v = result[0];
    std::cout << "Result: " << v << std::endl;
}
```

You can call `MATLABEngine::feval` using native C++ types. To do so, you must specify the returned type with the call to `MATLABEngine::feval` as:

```
feval<type>( ... )
```

For example, the returned type is `int` here:

```
int cresult = matlabPtr->feval<int>(u"gcd", 30, 56);
```

This example defines a `matlab::data::TypedArray` to pass an array of type `double` to the MATLAB `sqrt` function. Because one of the numbers in the array is negative, MATLAB returns a complex array as the result. Therefore, define the returned type as a `matlab::data::TypedArray<std::complex<double>>`.

```

#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void callFevalsqrt() {
    // Call MATLAB sqrt function on array

    using namespace matlab::engine;

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

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Define a four-element array
    matlab::data::TypedArray<double> const argArray =
        factory.createArray({ 1,4 }, { -2.0, 2.0, 6.0, 8.0 });

    // Call MATLAB function
    matlab::data::TypedArray<std::complex<double>> const results =
        matlabPtr->feval(u"sqrt", argArray);

    // Display results
    int i = 0;
    for (auto r : results) {
        double a = argArray[i++];
        double realPart = r.real();
        double imgPart = r.imag();
        std::cout << "Square root of " << a << " is " <<
            realPart << " + " << imgPart << "i" << std::endl;
    }
}

```

It is safe to use a `matlab::data::Array` for returned types when calling MATLAB functions. For example, you can write the previous example using a `matlab::data::Array` for the returned value.

```

void callFevalsqrt() {
    // Call MATLAB sqrt function on array

    using namespace matlab::engine;

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

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Define a four-element array
    matlab::data::Array const argArray =
        factory.createArray({ 1,4 }, { -2.0, 2.0, 6.0, 8.0 });

    // Call MATLAB function
    matlab::data::Array results = matlabPtr->feval(u"sqrt", argArray);

    // Display results
    for (int i = 0; i < results.getNumberOfElements(); i++) {
        double a = argArray[i];
        std::complex<double> v = results[i];
    }
}

```

```

double realPart = v.real();
double imgPart = v.imag();
std::cout << "Square root of " << a << " is " <<
    realPart << " + " << imgPart << std::endl;
}
}

```

Call Function with Name/Value Arguments

Some MATLAB functions accept optional name-value pair arguments. The names are character arrays and the values can be any type of value. Use a `std::vector` to create a vector of arguments containing the names and values in correct sequence.

This sample code calls the MATLAB `movsum` function to compute the three-point centered moving sum of a row vector, discarding endpoint calculations. This function call requires these arguments:

- Numeric array
- Scalar window length
- Name-value pair consisting of the character arrays `Endpoint` and `discard`

Here is the equivalent MATLAB code:

```

A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movsum(A,3,'Endpoints','discard');

```

Pass the arguments to `MATLABEngine::feval` as a `std::vector` containing these arguments for the MATLAB function. Create each argument using the `matlab::data::ArrayFactory`.

```

void callFevalmovsum() {
    //Pass vector containing various types of arguments

    using namespace matlab::engine;

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

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create a vector of input arguments
    std::vector<matlab::data::Array> args({
        factory.createArray<double>({ 1, 10 }), { 4, 8, 6, -1, -2, -3, -1, 3, 4, 5 }},
        factory.createScalar<int32_t>(3),
        factory.createCharArray("Endpoints"),
        factory.createCharArray("discard")
    });

    // Call MATLAB function

```



```

matlab::data::TypedArray<double> const result = matlabPtr->feval(u"movsum", args);

// Display results
int i = 0;
for (auto r : result) {
    std::cout << "results[" << i++ << "] = " << r << std::endl;
}
}

```

Call Function Asynchronously

This example calls the MATLAB `conv` function to multiply two polynomials. After calling `MATLABEngine::fevalAsync`, use `FutureResult::get` to get the result from MATLAB.

```

#include "MatlabdataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

static void callFevalAsync() {
    //Call MATLAB functions asynchronously

    using namespace matlab::engine;

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

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create input argument arrays
    std::vector<matlab::data::Array> args({
        factory.createArray<double>({ 1, 3 },{ 1, 0, 1 }},
        factory.createArray<double>({ 1, 2 },{ 2, 7 }
    });
    String func(u"conv");

    // Call function asynchronously
    FutureResult<matlab::data::Array> future = matlabPtr->fevalAsync(func, args);

    // Get results
    matlab::data::TypedArray<double> results = future.get();

    // Display results
    std::cout << "Coefficients: " << std::endl;
    for (auto r : results) {
        std::cout << r << " " << std::endl;
    }
}

```

Call Function with Multiple Returned Arguments

This sample code uses the MATLAB `gcd` function to find the greatest common divisor and Bézout coefficients from the two numeric values passes as inputs. The `gcd` function can

return either one or three arguments, depending on how many outputs the function call requests. In this example, the call to the MATLAB `gcd` function returns three outputs.

By default, `MATLABEngine::feval` assumes that the number of returned values is one. Therefore, you must specify the actual number of returned values as the second argument to `MATLABEngine::feval`.

In this example, `MATLABEngine::feval` returns a `std::vector` containing the three results of the `gcd` function call. The returned values are scalar integers.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void multiOutput() {
    //Pass vector containing MATLAB data array array

    using namespace matlab::engine;

    // Start MATLAB engine synchronously
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
    std::cout << "Started MATLAB Engine" << std::endl;

    //Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    //Create vector of MATLAB data array arrays
    std::vector<matlab::data::Array> args({
        factory.createScalar<int16_t>(30),
        factory.createScalar<int16_t>(56)
    });

    //Call gcd function, get 3 outputs
    const size_t numReturned = 3;
    std::vector<matlab::data::Array> result = matlabPtr->feval(u"gcd", numReturned, args);

    //Display results
    for (auto r : result) {
        std::cout << "gcd output: " << int16_t(r[0]) << std::endl;
    }
}
```

Call Function with Native C++ Types

You can use native C++ types when calling MATLAB functions. `MATLABEngine::feval` and `MATLABEngine::fevalAsync` accept certain scalar C++ types passed as MATLAB function arguments. To pass arrays and other types to MATLAB functions, use the MATLAB Data API. For more information on this API, see “MATLAB Data API”.

This example uses `int16_t` values as inputs and a `std::tuple` to return the results from the MATLAB `gcd` function.

Here is the equivalent MATLAB code.

```
[G,U,V] = gcd(int16(30),int16(56));

#include "MatlabEngine.hpp"
#include <iostream>
#include <tuple>

void multiOutputTuple() {
    //Return tuple from MATLAB function call

    using namespace matlab::engine;

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

    //Call MATLAB gcd function
    std::tuple<int16_t, int16_t, int16_t> nresults;
    nresults = matlabPtr->feval<std::tuple<int16_t, int16_t, int16_t>>
        (u"gcd", int16_t(30), int16_t(56));

    // Display results
    int16_t G;
    int16_t U;
    int16_t V;
    std::tie(G, U, V) = nresults;
    std::cout << "GCD : " << G << ", "
        << "Bezout U: " << U << ", "
        << "Bezout V: " << V << std::endl;
}
}
```

For specific information on member function syntax, see `matlab::engine::MATLABEngine`.

Control Number of Outputs

MATLAB functions can behave differently depending on the number of outputs requested. Some functions can return no outputs or a specified number of outputs.

For example, the MATLAB `pause` function holds execution for a specified number of seconds. However, if you call `pause` with an output argument, it returns immediately with a status value without pausing.

```
pause(20) % Pause for 20 seconds

state = pause(20); % No pause, return pause state
```

This example calls `pause` without assigning an output. With `void` output specified, MATLAB pauses execution for 20 seconds.

```
#include "MatlabEngine.hpp"
```

```
void voidOutput() {
    // No output from feval
    using namespace matlab::engine;

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

    // Call pause function with no output
    matlabPtr->feval<void>(u"pause", 20);
}
```

This call to `MATLABEngine::feval` uses the signature that defines the MATLAB function arguments as a `std::vector<matlab::data::Array>`. Without assigning an output argument, MATLAB pauses execution for 20 seconds.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"

void zeroOutput() {
    // No output from feval

    using namespace matlab::engine;

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

    //Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Call pause function with no output
    matlab::data::Array arg = factory.createScalar<int16_t>(20);
    const size_t numReturned = 0;
    matlabPtr->feval(u"pause", numReturned, { arg });
}
```

The MATLAB `clock` function returns the current date and time as a date vector. If you assign two outputs, `clock` returns the second output as a Boolean value indicating if it is Daylight Saving Time in the system time zone.

This example calls the `clock` function with one output or two outputs, depending on the value of an input argument. The second argument passed to the call to `MATLABEngine::feval` determines how many outputs to request from `clock`.

Call `MATLABEngine::feval` with these arguments.

Inputs

MATLAB function name	const matlab::engine::String
Number of outputs	const size_t
Input arguments for MATLAB function (empty)	std::vector<matlab::data::Array>

Outputs

All outputs	std::vector<matlab::data::Array>
-------------	----------------------------------

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void varOutputs(const bool tZone) {
    using namespace matlab::engine;

    // Start MATLAB engine synchronously
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
    std::cout << "Started MATLAB Engine" << std::endl;

    // Define number of outputs
    size_t numReturned(0);
    if (tZone) {
        numReturned = 2;
    } else {
        numReturned = 1;
    }
    std::vector<matlab::data::Array> dateTime = matlabPtr->feval(u"clock", numReturned, { });
    auto dateVector = dateTime[0];

    // Display results
    for (int i = 0; i < 6; i++) {
        std::cout << double(dateVector[i]) << " ";
    }

    if (tZone) {
        auto DTS = dateTime[1];
        if (bool(DTS[0])) {
            std::cout << "It is Daylight Saving Time" << std::endl;
        }
        else {
            std::cout << "It is Standard Time" << std::endl;
        }
    }
}
```

See Also

matlab::data::ArrayFactory | matlab::engine::MATLABEngine

Related Examples

- “Evaluate MATLAB Statements from C++” on page 14-29

Evaluate MATLAB Statements from C++

In this section...

“Evaluation of MATLAB Statements” on page 14-29

“Evaluate Mathematical Function in MATLAB” on page 14-29

Evaluation of MATLAB Statements

Evaluate MATLAB statements from C++ using the `MATLABEngine::eval` and `MATLABEngine::evalAsync` member functions. These member functions are similar to the MATLAB `eval` function. The `MATLABEngine::eval` and `MATLABEngine::evalAsync` functions do not return the results of evaluating the MATLAB statement.

Use `MATLABEngine::eval` and `MATLABEngine::evalAsync` when you do not need to pass arguments from C++ or return values to C++. The statements that you execute with these functions can access variables in the MATLAB workspace.

Here are some things to know about evaluating statements in MATLAB.

- These functions pass statements to MATLAB in as a `matlab::engine::String`.
- Convert an `std::string` to a `matlab::engine::String` using the `u"..."` literal or the utility function `matlab::engine::convertUTF8StringToUTF16String`.
- The input arguments named in the string must exist in the MATLAB workspace.
- You can assign the results of the evaluation to variables within the statement string. The variable that you assign in the statement is created in the MATLAB base workspace.
- MATLAB does not require you to initialize the variables created in the statement.
- You can store the standard output from MATLAB functions and error messages in stream buffers.

Evaluate Mathematical Function in MATLAB

This sample code uses `MATLABEngine::eval` to evaluate a series of MATLAB statements. These statements:

- Evaluate a mathematical function over a specified domain using `meshgrid` and `exp`.
- Create a graph of the function using `surf`.
- Export the graph to a JPEG file using `print`.

Here is the equivalent MATLAB code.

```
[X, Y] = meshgrid(-2:0.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(Z)
print('SurfaceGraph', '-djpeg')
currentFolder = pwd;
```

Here is the C++ code to execute these statements in MATLAB.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void evalSurfaceGraph() {
    // Evaluate functions in MATLAB

    using namespace matlab::engine;

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

    // Evaluate commands in MATLAB
    matlabPtr->eval(u"[X, Y] = meshgrid(-2:0.2:2);");
    matlabPtr->eval(u"Z = X .* exp(-X.^2 - Y.^2);");
    matlabPtr->eval(u"surf(Z)");
    matlabPtr->eval(u"print('SurfaceGraph', '-djpeg')");
    matlabPtr->eval(u"currentFolder = pwd;");

    // Get the name of the folder containing the jpeg file
    matlab::data::CharArray currentFolder = matlabPtr->getVariable(u"currentFolder");
    std::cout << "SurfaceGraph.jpg written to this folder: " <<
        currentFolder.toAscii() << std::endl;
}
```

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

See Also

`matlab::engine::MATLABEngine`

Related Examples

- “Call MATLAB Functions from C++” on page 14-19

Pass Variables from C++ to MATLAB

In this section...

“Ways to Pass Variables” on page 14-32

“Put Variables in MATLAB Base Workspace” on page 14-32

Ways to Pass Variables

You can pass C++ variables to MATLAB using these techniques:

- Pass the variables as function arguments in calls to the `matlab::engine::MATLABEngine` “`feval`” or “`fevalAsync`” member functions. Variables passed as arguments to function calls are not stored in the MATLAB base workspace. For more information, see “Call MATLAB Functions from C++” on page 14-19.
- Put the variables in the MATLAB base or global workspace using the `matlab::engine::MATLABEngine` “`setVariable`” and “`setVariableAsync`” member functions. For more information on using global variables in MATLAB, see the `MATLAB global` function.

You can create variables in the MATLAB workspace using the `matlab::engine::MATLABEngine` “`eval`” and “`evalAsync`” member functions. Use these functions to execute MATLAB statements that make assignments to variables. For more information, see “Evaluate MATLAB Statements from C++” on page 14-29.

Put Variables in MATLAB Base Workspace

This sample code performs these steps:

- Puts variables in the MATLAB workspace using `MATLABEngine::setVariable`
- Uses these variables to call the MATLAB `movsum` function using `MATLABEngine::eval`
- Gets the output variable `A` from the MATLAB workspace using `MATLABEngine::getVariable`.

Here is the equivalent MATLAB code.

```
A = movsum([4 8 6 -1 -2 -3 -1 3 4 5],3,'Endpoints','discard');
```

Here is the C++ code.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void callputVariables() {
    using namespace matlab::engine;

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

    //Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create variables
    auto data = factory.createArray<double>({ 1, 10 },
        { 4, 8, 6, -1, -2, -3, -1, 3, 4, 5 });
    auto windowLength = factory.createScalar<int32_t>(3);
    auto name = factory.createCharArray("Endpoints");
    auto value = factory.createCharArray("discard");

    // Put variables in the MATLAB workspace
    matlabPtr->setVariable(u"data", std::move(data));
    matlabPtr->setVariable(u"w", std::move(windowLength));
    matlabPtr->setVariable(u"n", std::move(name));
    matlabPtr->setVariable(u"v", std::move(value));

    // Call the MATLAB movsum function
    matlabPtr->eval(u"A = movsum(data, w, n, v);");

    // Get the result
    matlab::data::TypedArray<double> const A = matlabPtr->getVariable(u"A");

    // Display the result
    int i = 0;
    for (auto r : A) {
        std::cout << "results[" << i << "] = " << r << std::endl;
        ++i;
    }
}
```

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

See Also

matlab::engine::MATLABEngine | matlab::engine::startMATLAB

Related Examples

- “Pass Variables from MATLAB to C++” on page 14-34

Pass Variables from MATLAB to C++

In this section...

“Bring Result of MATLAB Calculation Into C++” on page 14-34

“Get MATLAB Objects and Access Properties” on page 14-35

“Set Property on MATLAB Object” on page 14-35

Pass variables from the MATLAB base or global workspace to your C++ program using the `matlab::engine::MATLABEngine` “`getVariable`” and “`getVariableAsync`” member functions. Return the variable to C++ as a `matlab::data::Array`.

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Bring Result of MATLAB Calculation Into C++

This sample code performs a calculation in MATLAB using `MATLABEngine::eval` and gets the results using `MATLABEngine::getVariable`.

The MATLAB `cart2sph` function converts a point in Cartesian coordinates to its representation in spherical coordinates.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void callgetVars() {
    using namespace matlab::engine;

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

    // Evaluate MATLAB statement
    matlabPtr->eval(u"az,el,r] = cart2sph(5,7,3);");

    // Get the result from MATLAB
    matlab::data::TypedArray<double> result1 = matlabPtr->getVariable(u"az");
    matlab::data::TypedArray<double> result2 = matlabPtr->getVariable(u"el");
    matlab::data::TypedArray<double> result3 = matlabPtr->getVariable(u"r");

    // Display results
    std::cout << "az: " << result1[0] << std::endl;
    std::cout << "el: " << result2[0] << std::endl;
    std::cout << "r: " << result3[0] << std::endl;
}
```

Get MATLAB Objects and Access Properties

Use the `matlab::engine::MATLABEngine` “`getVariable`” or “`getVariableAsync`” member functions to get MATLAB object variables. Return the object to C++ as a `matlab::data::Array`. Access object properties using the `matlab::engine::MATLABEngine` “`getProperty`” or “`getPropertyAsync`” member functions.

This sample code creates a MATLAB figure object and returns the object handle to C++. Values of the figure `Units` property are always character arrays. Therefore, to query the figure `Units` property, return a `matlab::data::CharArray` with the value of the figure property (default value is `pixels`).

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void callgetVariables() {
    using namespace matlab::engine;

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

    // Create figure window
    matlabPtr->eval(u"figureHandle = figure;");

    //Get figure handle and Units property
    matlab::data::Array figHandle = matlabPtr->getVariable(u"figureHandle");
    matlab::data::CharArray units = matlabPtr->getProperty(figHandle, u"Units");

    // Display property value
    std::cout << "Units property: " << units.toAscii() << std::endl;
}
```

Get Property from Object Array

If the object variable is an array of objects, call `getProperty` or `getPropertyAsync` with the index of the object in the array that you want to access. For example, to get the value of the `Units` property of the fourth element in the object array, `objectArray`, specify the index as the second input argument.

```
matlab::data::CharArray units = matlabPtr->getProperty(objectArray, 3, u"Units");
```

Set Property on MATLAB Object

To set the value of a MATLAB object property from C++, use the `matlab::engine::MATLABEngine` “`setProperty`” or “`setPropertyAsync`” member function. To access the object property, you can get the object variable from the MATLAB

workspace and set the value on the C++ variable. The property value that you set updates the object in the MATLAB workspace.

You can also return an object from a call to `MATLABEngine::feval` and `MATLABEngine::fevalAsync` and set property values on that object.

This sample code creates a MATLAB figure object and returns the object to C++. The code sets the figure `Color` property to `red`, which changes the value of the property on the object in the MATLAB workspace.

Because the value of the `Color` property is a char array, use the `matlab::data::ArrayFactory` to create a `matlab::data::CharArray` to define the new value.

```
void getObject() {
    using namespace matlab::engine;

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

    //Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create figure window
    size_t numArguments(1);
    std::vector<matlab::data::Array> figureHandle = matlabPtr->feval(u"figure", numArguments, {});

    // Pause to display the figure
    matlabPtr->eval(u"pause(5)");

    // Set the Color property to red
    matlabPtr->setProperty(figureHandle[0], u"Color", factory.createCharArray("red"));
    matlabPtr->eval(u"pause(10)");
}
```

Set Property from Object Array

If the object variable is an array of objects, call `setProperty` or `setPropertyAsync` with the index of the object in the array that you want to access. For example, to set the value of the `Color` property of the fourth element in the object array, `objectArray`, specify the index as the second input argument.

```
matlabPtr->setProperty(objectArray, 3, u"Color", factory.createCharArray("red"));
```

See Also

`matlab::engine::MATLABEngine` | `matlab::engine::startMATLAB`

Related Examples

- “Pass Variables from C++ to MATLAB” on page 14-32

Redirect MATLAB Command Window Output to C++

In this section...

“Redirect Screen Output” on page 14-38

“Redirect Error Output” on page 14-39

MATLAB displays error messages and the output from statements in the MATLAB command window. To redirect this output to your C++ program, use a string buffer to capture this output and return it with the “feval”, “fevalAsync”, “eval”, or “evalAsync” member function.

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Redirect Screen Output

This sample code evaluates two statements in MATLAB. These statements create three variables in the MATLAB workspace. The code calls the MATLAB whos function, which displays the current workspace variables in the MATLAB command window. Capture MATLAB standard output in a string buffer by passing a pointer to the buffer with the call to `MATLABEngine::eval`.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void screenOutput() {
    using namespace matlab::engine;

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

    // Evaluate statements that create variables
    matlabPtr->eval(u"[X,Y] = meshgrid(-2:.2:2);");
    matlabPtr->eval(u"Z = X.*exp(-X.^2 - Y.^2);");

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create string buffer for standard output
    typedef std::basic_stringbuf<char16_t> StringBuffer;
    auto output = std::make_shared<StringBuf>();

    // Display variables in the MATLAB workspace
    matlabPtr->eval(u"whos", output);
}
```



```

// Display MATLAB output in C++
String output_ = output.get()->str();
std::cout << convertUTF16StringToUTF8String(output_) << std::endl;
}

```

Redirect Error Output

This sample code causes a MATLAB error by referencing a variable after clearing all variables from the MATLAB workspace. The string buffer passed to the `MATLABEngine::eval` member function captures the error message inside a try/catch code block.

```

#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void errorOutput() {
    using namespace matlab::engine;

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

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create string buffer for standard output
    typedef std::basic_stringbuf<char16_t> StringBuf;
    auto error = std::make_shared<StringBuf>();

    // Evaluate statement that causes error
    matlabPtr->eval(u"clear");
    try {
        matlabPtr->eval(u"x + 2;", {}, error);
    }
    catch (...) {
        String error_ = error.get()->str();
        std::cout << convertUTF16StringToUTF8String(error_) << std::endl;
    }
}

```

See Also

[matlab::data::ArrayFactory](#) | [matlab::engine::MATLABEngine](#) |
[matlab::engine::convertUTF16StringToUTF8String](#) |
[matlab::engine::startMATLAB](#)

Related Examples

- “Evaluate MATLAB Statements from C++” on page 14-29

Create Cell Arrays from C++

In this section...

“Put Cell Array in MATLAB Workspace” on page 14-41

“Access Elements of Cell Array” on page 14-42

“Modify Elements of Cell Array” on page 14-42

A MATLAB cell array is a container in which each cell can contain an array of any type. The MATLAB C++ engine enables you to create cell arrays using the `matlab::data::ArrayFactory::createCellArray` member function. To create an empty cell array, use `createArray<matlab::data::Array>`.

You can pass cell arrays to and from MATLAB. In MATLAB, these arrays are of class `cell`.

The `matlab::data::CellArray` class is implemented as an array of arrays defined as:

```
matlab::data::TypedArray<matlab::data::Array>
```

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Put Cell Array in MATLAB Workspace

This sample code creates a cell array and puts it in the MATLAB base workspace.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"

void cellArrayPut() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create cell array
    matlab::data::CellArray cellArray1 = factory.createCellArray({ 1,2 },
        factory.createCharArray("MATLAB Cell Array"),
        factory.createArray<double>({ 2,2 }, { 1.2, 2.2, 3.2, 4.2 }));

    // Put cell array in MATLAB workspace
```

```
    matlabPtr->setVariable(u"cellArray1", cellArray1);  
}
```

Calling the MATLAB whos function shows the variable in the MATLAB workspace.

```
>> whos  
Name           Size           Bytes  Class  Attributes  
  
cellArray1     1x2             290   cell  
>> cellArray1{:}  
  
ans =  
  
    'MATLAB Cell Array'  
  
ans =  
  
    1.2000    3.2000  
    2.2000    4.2000
```

Access Elements of Cell Array

Use [] indexing to access the elements of a cell array. For example, access the first element of the cell array created in the previous section and convert the element to a `std::string`:

```
matlab::data::CharArray cArray = cellArray1[0][0];  
std::string str = cArray.toAscii();
```

The variable `str` is a copy of the value in the cell array.

Modify Elements of Cell Array

There are different ways to modify elements in a cell array:

- Create a reference to the element and modify the value in the cell array.
- Copy the element, change the value, then reassign the value to the cell array.

This sample code uses `matlab::data::TypedArrayRef` to create a reference to the array contained in a specific cell. Then the array is modified by changing specific elements in the array using indexed assignment.

```

#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"

void cellArrayMod() {
    //Modify elements of a cell array

    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create a 2-by-2 array of arrays
    matlab::data::CellArray cellArray2 = factory.createArray<matlab::data::Array>({ 2,2 });

    // Assign values to each cell
    cellArray2[0][0] = factory.createCharArray("A cell array");
    cellArray2[0][1] = factory.createArray<double>({ 1,3 }, { 2.2, 3.2, -4.2 });
    cellArray2[1][0] = factory.createArray<bool>({ 1,3 }, { true, true, false });
    cellArray2[1][1] = factory.createScalar<int32_t>(-3374);

    // Get reference to elements of the cell array
    // Modify the elements in the cell array
    matlab::data::TypedArrayRef<double> elem1 = cellArray2[0][1];
    elem1[1] = -3.2;
    matlab::data::TypedArrayRef<bool> elem2 = cellArray2[1][0];
    elem2[1] = false;

    // Put cell array in MATLAB workspace
    matlabPtr->setVariable(u"cellArray2", std::move(cellArray2));
}

```

The cell array in the MATLAB workspace includes the changes made to the array element references. Here is the cell array in MATLAB.

```

>> cellArray2{:}

ans =

    'A cell array'

ans =

    1×3 logical array

    1    0    0

```

```
ans =  
    2.2000   -3.2000   -4.2000
```

```
ans =  
int32  
    -3374
```

Get Cell Array from MATLAB

This sample code gets a cell array from MATLAB. This code assumes that there is a cell array variable named `cellArray2` in the MATLAB workspace, like the one created in the previous example. To pass the cell array to MATLAB, see “Modify Elements of Cell Array” on page 14-42.

To get the cell array, follow these steps:

- Use the `matlab::engine::MATLABEngine` “`getVariable`” member function to bring the cell array into C++.
- Define the returned cell array as a `matlab::data::CellArray`.
- Return the contents of the cell indexed by `[1][0]` as a `matlab::data::TypedArray<bool>`.
- Use the `TypedArray::getNumberOfElements` member function to loop through the elements of the array cell.

```
void cellArrayGet() {  
    using namespace matlab::engine;  
  
    // Connect to named shared MATLAB session started as:  
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"  
    String session(u"myMatlabEngine");  
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);  
  
    // Get a cell array from MATLAB  
    matlab::data::CellArray ca = matlabPtr->getVariable(u"cellArray2");  
  
    // Copy elements of a cell into vector  
    matlab::data::TypedArray<bool> const elem2 = ca[1][0];  
    std::vector<bool> logicalCell(elem2.getNumberOfElements());  
    int i = 0;  
    for (auto e : elem2) {  
        logicalCell[i] = e;  
        ++i;  
    }  
}
```

```
}  
}
```

See Also

`matlab::data::ArrayFactory` | `matlab::data::TypedArray` |
`matlab::engine::MATLABEngine` | `matlab::engine::connectMATLAB`

Related Examples

- “Connect C++ to Running MATLAB Session” on page 14-16
- “Operate on C++ Arrays Using Visitor Pattern” on page 2-11

Create Structure Arrays from C++

In this section...

“Create Structure Array and Send to MATLAB” on page 14-46

“Get Structure from MATLAB” on page 14-47

“Access Struct Array Data” on page 14-47

MATLAB structures contain data that you reference with field names. Each field can contain any type of data. To access data in a structure, MATLAB code uses dot notation of the form `structName.fieldName`. The class of a MATLAB structure is `struct`.

In an array of MATLAB structures, each structure must have the same field names.

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

Create Structure Array and Send to MATLAB

This sample code creates a structure array and puts it in the MATLAB workspace.

Here is how to create and send the array.

- Create an empty `matlab::data::StructArray`, defining the array dimensions and the field names.
- Assign values to the fields using array and field name indices. Define the correct array type using the `matlab::data::ArrayFactory`.
- Put the structure array in the MATLAB workspace using the `MATLABEngine::setVariable` member function.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void putStructArray() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create MATLAB data array factory
```



```

matlab::data::ArrayFactory factory;

// Define 2-element struct array with two fields per struct
auto structArray = factory.createStructArray({ 1, 2}, { "f1", "f2" });

// Assign values to each field in first struct
structArray[0]["f1"] = factory.createCharArray("First Data Set");
structArray[0]["f2"] = factory.createArray<uint8_t>({ 1, 3 }, { 1, 2, 3 });

// Assign values to each field in second struct
structArray[1]["f1"] = factory.createCharArray("Second Data Set");
structArray[1]["f2"] = factory.createArray<double>({ 1, 5 }, { 4., 5., 6., 7., 8. });

// Put struct array in MATLAB workspace
matlabPtr->setVariable(u"structArray", structArray);
}

```

Get Structure from MATLAB

Get a structure variable from the MATLAB workspace using the `matlab::engine::MATLABEngine` “`getVariable`” member function.

Note This sample code gets a structure array from the MATLAB workspace. This code assumes that there is a structure array variable named `structArray` in the MATLAB workspace, like the one created in the previous example. To pass the structure array to MATLAB, see “Create Structure Array and Send to MATLAB” on page 14-46.

```

#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"

void readStructArray() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Get the struct array from MATLAB
    matlab::data::StructArray matlabStruct = matlabPtr->getVariable(u"structArray");
}

```

Access Struct Array Data

There are different ways to access the structure in C++:

- Create a reference to a particular field. Changes to the reference modify the value in the structure.

- Create a copy of the field values. Changes to the copy do not modify the values in the structure unless you reassign the value to the structure field.

To get information about the structure array, use the `matlab::data::StructArray` member functions `getDimensions`, `getNumberOfFields`, and `getFieldNames`

This sample code follows these steps:

- Gets the structure array variable named `structArray` from the MATLAB session.
- Creates a reference to one of the structure fields.
- Modifies an element of the double array contained in the field using the reference.
- Returns the modified structure array to the shared MATLAB session.

This sample code gets the structure array from the shared MATLAB session that was created in a previous section, “Create Structure Array and Send to MATLAB” on page 14-46.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void modifyStructArray() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Get the struct array from MATLAB
    matlab::data::StructArray matlabStruct = matlabPtr->getVariable(u"structArray");
    auto numelem = matlabStruct.getDimensions();
    std::cout << "structArray size is: " << numelem[0] << " by " << numelem[1] << std::endl;

    // Get number of fields
    size_t numFields = matlabStruct.getNumberOfFields();
    std::cout << "structArray has " << numFields << " fields" << std::endl;

    // Get the struct array fieldnames
    auto fields = matlabStruct.getFieldNames();
    std::vector<matlab::data::MATLABFieldIdentifier> fieldNames;
    for (const auto& name : fields) {
        fieldNames.push_back(name);
    }

    // Change value of array element using a reference
    matlab::data::TypedArrayRef<double> field1 = matlabStruct[1][fieldNames[1]];
    field1[0] = -200.;
}
```

```
// Return modified struct array to MATLAB
matlabPtr->setVariable(u"structArray", matlabStruct);
}
```

See Also

[matlab::data::ArrayFactory](#) | [matlab::data::MATLABFieldIdentifier](#) |
[matlab::data::StructArray](#) | [matlab::engine::MATLABEngine](#)

Related Examples

- “Create Cell Arrays from C++” on page 14-41

Pass Enumerations to MATLAB From C++

To call MATLAB functions that require MATLAB enumeration members as inputs, define the enumeration member as a `matlab::data::EnumArray`. Use `matlab::data::ArrayFactory` to create the enumeration array. The `matlab::data::EnumArray` contains the MATLAB class name and one or more enumeration members. You can also pass the array as a variable to the MATLAB workspace using `MATLABEngine::setVariable`.

Note To pass a `matlab::data::EnumArray` to MATLAB, the named MATLAB class must exist and be on the MATLAB path.

Suppose that you define the following `TextString` in MATLAB. This class defines a property that is typed as a specific enumeration class named `TextColor`. The `TextString` class constructor takes two input arguments:

- `Str` — A 1-by-n char array
- `Color` — An enumeration member of the `TextColor` class.

```
classdef TextString
    properties
        Str(1,:) char
        Color TextColor
    end
    methods
        function obj = TextString(str,color)
            if nargin == 2
                obj.Str = str;
                obj.Color = color;
            end
        end
    end
end
```

Here is how to define the MATLAB `TextColor` enumeration class.

```
classdef TextColor
    enumeration
        Red
        Green
        Blue
    end
end
```

```

    end
end

```

This MATLAB statement creates a `TextString` object by passing a character vector and an enumeration member to the class constructor.

```
T = TextString('Any text string',TextColor.Blue);
```

The following sample code creates a MATLAB `TextString` object and displays the property values. To create the `TextString` object:

- Define a `matlab::data::CharArray` for the MATLAB character vector argument.
- Define a `matlab::data::EnumArray` for the MATLAB `TextColor.Blue` enumeration argument.
- Pass the argument vector to `MATLABEngine::feval`.
- Get the property values using `MATLABEngine::getProperty` and display the values.

Note This example requires you to define the MATLAB `TextString` and `TextColor` classes described here. These classes must be on the path of the shared MATLAB session used by this example.

```

#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

void enumArray() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create enumeration array
    auto enumColor = factory.createEnumArray({ 1,1 }, "TextColor", { "Blue" });

    // Create argument vector
    std::vector<matlab::data::Array> args({
        factory.createCharArray("Any text string"),
        enumColor});

    // Call MATLAB TextString to create object
    matlab::data::Array T = matlabPtr->feval(u"TextString", args);

    // Get the value of the Str property

```

```
matlab::data::CharArray c = matlabPtr->getProperty(T, u"Str");
std::cout << "Str property value: " << c.toAscii() << std::endl;

// Get the value of the Color property
matlab::data::EnumArray col = matlabPtr->getProperty(T, u"Color");
std::cout << "Color property class: " << col.getClassName() << std::endl;
std::cout << "Color property value: " << std::string(col[0]) << std::endl;
}
```

Here is the program output.

```
Str property value: Any text string
Color property class: TextColor
Color property value: Blue
```

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

See Also

matlab::data::ArrayFactory | matlab::engine::MATLABEngine |
matlab::engine::connectMATLAB

Related Examples

- “Call MATLAB Functions from C++” on page 14-19

Pass Sparse Arrays to MATLAB From C++

MATLAB sparse arrays provide efficient storage of double or logical data that has a large percentage of zeros. MATLAB sparse arrays support arithmetic, logical, and indexing operations. For more information, see “Sparse Matrices”.

Use `matlab::data::ArrayFactory` to create a `matlab::data::SparseArray` array. Write the data for the sparse array into buffers and use these buffers to create the sparse array. Pass the sparse array to MATLAB using `MATLABEngine::setVariable`.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"

void sparseArray() {
    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Define the data for the sparse array
    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;

    // Create buffers for the data
    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);

    // Write data into the buffers
    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; });

    // Use the buffers to create the sparse array
    matlab::data::SparseArray<double> arr =
        factory.createSparseArray<double>({ 2,9 }, nnz,
            std::move(data_p), std::move(rows_p), std::move(cols_p));

    // Put the sparse array in the MATLAB workspace
    matlabPtr->setVariable(u"s", arr);
}
```

The MATLAB `whos` command shows that the array passed to the MATLAB workspace is a sparse array.

```
>> whos
Name          Size          Bytes Class          Attributes
```

s 2x9 128 double sparse

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

See Also

`matlab::data::ArrayFactory` | `matlab::engine::MATLABEngine` |
`matlab::engine::connectMATLAB`

Run Simulink Simulation from C++

In this section...

“MATLAB Command to Run Simulation” on page 14-55

“Simulink vdp Model from C++” on page 14-55

“MATLAB Code to Run Simulation” on page 14-56

“C++ Code to Run Simulation” on page 14-57

MATLAB Command to Run Simulation

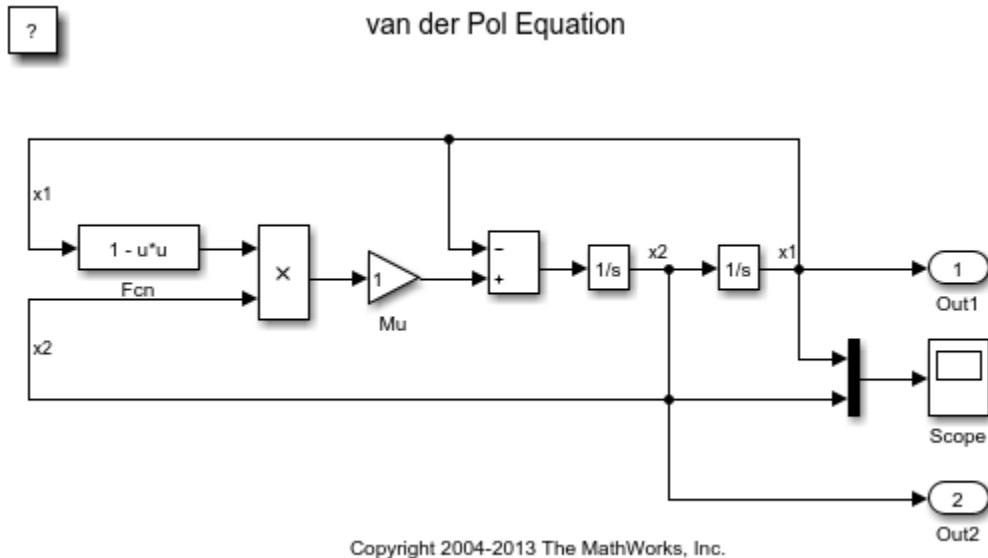
You can run Simulink simulations using the MATLAB Engine API for C++. Here are the basic steps to run a simulation programmatically:

- Start a MATLAB session.
- Load the Simulink model in MATLAB using the MATLAB `load_system` function.
- Run the simulation with specific simulation parameters using the MATLAB `sim` function.
- Access the results of the simulation using methods of the returned `Simulink.SimulationOutput` object.

For information on running simulations programmatically from MATLAB, see “Run Individual Simulations” (Simulink).

Simulink vdp Model from C++

The Simulink vdp block diagram simulates the van der Pol equation, which is a second order differential equation. Simulink solves the equations using the initial conditions and configuration parameters defined by the model.



MATLAB Code to Run Simulation

This MATLAB code shows the commands to run the simulation programmatically. The `Simulink.SimulationOutput` object `get` method returns the results and time vector.

```
load_system('vdp');
parameterStruct.SaveOutput = 'on';
parameterStruct.OutputSaveName = 'yOut';
parameterStruct.SaveTime = 'on';
parameterStruct.TimeSaveName = 'tOut';
simOut = sim('vdp',parameterStruct);
y = simOut.get('yOut');
t = simOut.get('tOut');
```

This MATLAB code creates a graph of the simulation output and exports the graph to a JPEG image file.

```
plot(t,y)
print('vdpSimulation','-djpeg')
```

C++ Code to Run Simulation

This sample code runs the Simulink simulation for the vdp model. The code performs these operations:

- Connect to a named MATLAB session that has Simulink installed.
- Create a `matlab::data::ArrayFactory` and build a `StructArray` that contains the simulation parameters.
- Pass the simulation parameter structure to the MATLAB workspace.
- Load the vdp Simulink model using an asynchronous call to `MATLABEngine::evalAsync`. Execution waits for MATLAB to load the model.
- Run the simulation using another asynchronous call to `MATLABEngine::evalAsync`. Execution waits for the simulation to finish.
- Extract the simulation data from the `Simulink.SimulationOutput` object in the MATLAB workspace.
- Graph the data and export a JPEG image to the MATLAB current folder using `MATLABEngine::eval` to execute MATLAB commands.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <chrono>
#include <iostream>

void runSimulation() {

    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create MATLAB data array factory
    matlab::data::ArrayFactory factory;

    // Create struct for simulation parameters
    auto parameterStruct = factory.createStructArray({ 1,4 }, {
        "SaveOutput",
        "OutputSaveName",
        "SaveTime",
        "TimeSaveName" });
    parameterStruct[0]["SaveOutput"] = factory.createCharArray("on");
    parameterStruct[0]["OutputSaveName"] = factory.createCharArray("yOut");
    parameterStruct[0]["SaveTime"] = factory.createCharArray("on");
    parameterStruct[0]["TimeSaveName"] = factory.createCharArray("tOut");

    // Put simulation parameter struct in MATLAB
    matlabPtr->setVariable(u"parameterStruct", parameterStruct);

    // Load vdp Simulink model
```

```
FutureResult<void> loadFuture = matlabPtr->evalAsync(u"load_system('vdp')");
std::cout << "Loading Simulink model... " << std::endl;
std::future_status loadStatus;
do {
    loadStatus = loadFuture.wait_for(std::chrono::seconds(1));
} while (loadStatus != std::future_status::ready);
std::cout << "vdp model loaded\n";

// Run simulation
FutureResult<void> simFuture = matlabPtr->evalAsync(u"simOut = sim('vdp',parameterStruct);");
std::cout << "Running simulation... " << std::endl;
std::future_status simStatus;
do {
    simStatus = loadFuture.wait_for(std::chrono::seconds(1));
} while (simStatus != std::future_status::ready);
std::cout << "vdp simulation complete\n";

// Get simulation data and create a graph
matlabPtr->eval(u"y = simOut.get('yOut');");
matlabPtr->eval(u"t = simOut.get('tOut');");
matlabPtr->eval(u"plot(t,y)");
matlabPtr->eval(u"pause(10)");
matlabPtr->eval(u"print('vdpSimulation','-djpeg')");
}
```

For information on how to setup and build C++ engine programs, see “Build C++ Engine Programs” on page 14-9.

See Also

`matlab::data::ArrayFactory` | `matlab::engine::MATLABEngine` |
`matlab::engine::connectMATLAB`

Related Examples

- “Evaluate MATLAB Statements from C++” on page 14-29
- “Save and Load Workspace Variables”

Convert C++ Engine Application to MATLAB Compiler SDK Application

You can deploy an engine application by modifying your code to use the MATLAB Compiler SDK API. For information about this API, see “C++ MATLAB Data API Shared Library Integration” (MATLAB Compiler SDK). Make these modifications to use the equivalent Compiler SDK C++ values:

- Header file

```
MatlabCppSharedLib.hpp
```

- Namespace

```
matlab::cpplib
```

- Class name

```
matlab::cpplib::MATLABApplication
```

- Compiler SDK requires the path of CTF (library archive file) to be set. For example:

```
const std::u16string U16STR_CTF_NAME = u"libtriangle.ctf";
auto lib = mc::initMatlabLibrary(app, U16STR_CTF_NAME);
```

Applications to Call Sierpinski Function

This example is an engine application that calls a MATLAB function `sierpinski` which calculates the points in Sierpinski's triangle. The example shows how to modify the engine code to run with MATLAB Compiler SDK.

If you have Compiler SDK, then you can use the graphical Sierpinski function `sierpinski.m` in the `matlabroot/extern/examples/compiler/sdk/c_cpp/triangle` folder.

Copy this code into the file `triangleEngine.cpp`.

Engine Application `triangleEngine.cpp`

```
/*=====
 *
 * triangleEngine.cpp
 *
 * MATLAB Engine API for C++ application
 *=====*/

#include "MatlabEngine.hpp"
#include <iostream>
```

```

namespace mc = matlab::engine;
namespace md = matlab::data;

const int DEFAULT_NUMBER_OF_POINTS = 1000;
std::u16string convertAsciiToUtf16(const std::string & asciiStr);

int mainFunc(std::shared_ptr<mc::MATLABEngine> app,
const int argc, const char * argv[]) {
    try {
        size_t numPoints(DEFAULT_NUMBER_OF_POINTS);
        if(argc > 1) {
            numPoints = std::stoi(argv[1]);
        }

        // Create arguments as input of the MATLAB function
        md::ArrayFactory factory;
        auto numPointsAsArray = factory.createScalar(numPoints);
        auto doDrawAsArray = factory.createScalar(doDraw); // create a logical scalar
        std::vector<md::Array> inputs{numPointsAsArray, doDrawAsArray};

        // The Sierpinski function returns the X and Y coordinates of the points
        // in the triangle. If doDraw is true, it also draws the figure.
        app->feval(u"sierpinski", 2, inputs);

        // Wait until the user closes the figure.
        app->eval(u"figures = findobj(allchild(groot),'flat','type','figure','visible','on');
            waitfor(figures(1))");

    } catch(std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return 0;
}

int main(const int argc, const char * argv[]) {
    int ret = 0;
    try {
        std::vector<std::u16string> options = {};
        auto matlabApplication = mc::connectMATLAB();//startMATLAB();
        ret = mainFunc(std::move(matlabApplication), argc, argv);
    } catch(const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return ret;
}

```

Modify the code to use the equivalent Compiler SDK C++ values.

If you copy the following code into `triangleSDK.cpp` and compare the file with `triangleEngine.cpp`, then you can see other modifications relevant to calling the Sierpinski function.

Corresponding Deployed Application `triangleSDK.cpp`

```

/*=====
*

```

```

* triangleSDK.cpp
* Use the generic interface for the C++ shared library
* generated by MATLAB Compiler SDK.
*=====*/

#include "MatlabCppSharedLib.hpp"
#include <iomanip>
#include <iostream>

namespace mc = matlab::cpplib;
namespace md = matlab::data;

const int DEFAULT_NUMBER_OF_POINTS = 1000;

int mainFunc(std::shared_ptr<mc::MATLABApplication> app,
const int argc, const char * argv[]) {
    try {
        size_t numPoints(DEFAULT_NUMBER_OF_POINTS);
        if(argc > 1) {
            numPoints = std::stoi(argv[1]);
        }
        // Pass the path of the CTF (library archive file) to initMATLAB library
        const std::u16string U16STR_CTF_NAME = u"libtriangle.ctf";
        auto lib = mc::initMATLABLibrary(app, U16STR_CTF_NAME);

        // Create arguments as input of the MATLAB function
        md::ArrayFactory factory;
        auto numPointsAsArray = factory.createScalar(numPoints);
        auto doDrawAsArray = factory.createScalar(doDraw); // create a logical scalar
        std::vector<md::Array> inputs{numPointsAsArray, doDrawAsArray};

        // The Sierpinski function returns the X and Y coordinates of the points
        // in the triangle. If doDraw is true, it also draws the figure.
        lib->feval(u"sierpinski", 2, inputs);

        // Wait until the user closes the figure.
        lib->waitForFiguresToClose();

    } catch(std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return 0;
}

int main(const int argc, const char * argv[]) {
    int ret = 0;
    try {
        auto mode = mc::MATLABApplicationMode::IN_PROCESS;
        std::vector<std::u16string> options = {};
        auto matlabApplication = mc::initMATLABApplication(mode, options);
        ret = mc::runMain(mainFunc, std::move(matlabApplication), argc, argv);
    } catch(const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return ret;
}

```

See Also

More About

- “C++ MATLAB Data API Shared Library Integration” (MATLAB Compiler SDK)
- “Build C++ Engine Programs” on page 14-9

Using .NET Libraries from MATLAB

- “Read Cell Arrays of Excel Spreadsheet Data” on page 15-4
- “Access a Simple .NET Class” on page 15-6
- “Load a Global .NET Assembly” on page 15-12
- “Work with Microsoft Excel Spreadsheets Using .NET” on page 15-13
- “Work with Microsoft Word Documents Using .NET” on page 15-15
- “Assembly Is Library of .NET Classes” on page 15-16
- “Limitations to .NET Support” on page 15-17
- “System Requirements for Using MATLAB Interface to .NET” on page 15-19
- “Using .NET from MATLAB” on page 15-20
- “Using a .NET Object” on page 15-22
- “Build a .NET Application for MATLAB Examples” on page 15-24
- “Troubleshooting Security Policy Settings from Network Drives” on page 15-25
- “.NET Terminology” on page 15-26
- “Simplify .NET Class Names” on page 15-27
- “Use import in MATLAB Functions” on page 15-28
- “Nested Classes” on page 15-29
- “Handle .NET Exceptions” on page 15-30
- “Pass Numeric Arguments” on page 15-31
- “Pass System.String Arguments” on page 15-32
- “Pass System.Enum Arguments” on page 15-34
- “Pass System.Nullable Arguments” on page 15-36
- “Pass Cell Arrays of .NET Data” on page 15-40
- “Pass Jagged Arrays” on page 15-43
- “Convert Nested System.Object Arrays” on page 15-47
- “Pass Data to .NET Objects” on page 15-49
- “Handle Data Returned from .NET Objects” on page 15-57

- “Use Arrays with .NET Applications” on page 15-63
- “Convert .NET Arrays to Cell Arrays” on page 15-66
- “Limitations to Support of .NET Arrays” on page 15-69
- “Set Static .NET Properties” on page 15-70
- “Using .NET Properties” on page 15-72
- “MATLAB Does Not Display Protected Properties” on page 15-74
- “Work with .NET Methods Having Multiple Signatures” on page 15-75
- “Call .NET Methods With out Keyword” on page 15-77
- “Call .NET Methods With ref Keyword” on page 15-79
- “Call .NET Methods With params Keyword” on page 15-81
- “Call .NET Methods with Optional Arguments” on page 15-83
- “Calling .NET Methods” on page 15-86
- “Calling .NET Methods with Optional Arguments” on page 15-88
- “Calling .NET Extension Methods” on page 15-89
- “Call .NET Properties That Take an Argument” on page 15-90
- “How MATLAB Represents .NET Operators” on page 15-92
- “Limitations to Support of .NET Methods” on page 15-93
- “Use .NET Events in MATLAB” on page 15-94
- “Call .NET Delegates in MATLAB” on page 15-97
- “Create Delegates from .NET Object Methods” on page 15-99
- “Create Delegate Instances Bound to .NET Methods” on page 15-101
- “.NET Delegates With out and ref Type Arguments” on page 15-103
- “Combine and Remove .NET Delegates” on page 15-104
- “.NET Delegates” on page 15-106
- “Calling .NET Methods Asynchronously” on page 15-107
- “Limitations to Support of .NET Events” on page 15-108
- “Limitations to Support of .NET Delegates” on page 15-109
- “Use Bit Flags with .NET Enumerations” on page 15-110
- “Read Special System Folder Path” on page 15-114
- “Default Methods for an Enumeration” on page 15-116

- “NetDocEnum Example Assembly” on page 15-118
- “Work with Members of a .NET Enumeration” on page 15-119
- “Refer to a .NET Enumeration Member” on page 15-121
- “Display .NET Enumeration Members as Character Vectors” on page 15-123
- “Convert .NET Enumeration Values to Type Double” on page 15-124
- “Iterate Through a .NET Enumeration” on page 15-125
- “Use .NET Enumerations to Test for Conditions” on page 15-127
- “Underlying Enumeration Values” on page 15-129
- “Limitations to Support of .NET Enumerations” on page 15-130
- “Create .NET Collections” on page 15-131
- “Convert .NET Collections to MATLAB Arrays” on page 15-133
- “Create .NET Arrays of Generic Type” on page 15-134
- “Display .NET Generic Methods Using Reflection” on page 15-135
- “.NET Generic Classes” on page 15-138
- “Accessing Items in .NET Collections” on page 15-139
- “Call .NET Generic Methods” on page 15-140

Read Cell Arrays of Excel Spreadsheet Data

This example shows how to convert columns of Microsoft Excel® spreadsheet data to MATLAB types. MATLAB reads a range of .NET values as a `System.Object[,]` type. Use the `cell` function to convert `System.String` values to MATLAB character arrays and `System.DateTime` values to `datetime` objects.

Create a file in Excel that contains the following data.

Date	Weight
10/31/96	174.8
11/29/96	179.3
12/30/96	190.4
01/31/97	185.7

Right-click the **Date** column, select **Format Cells**, and then the **Number** tab. Verify that the value for **Category:** is **Date**.

Name the file `weight.xls` in the `H:\Documents\MATLAB` folder. Close the file.

In MATLAB, read the data from the spreadsheet.

```
NET.addAssembly('microsoft.office.interop.excel');  
app = Microsoft.Office.Interop.Excel.ApplicationClass;  
book = app.Workbooks.Open('H:\Documents\MATLAB\weight.xls');  
sheet = Microsoft.Office.Interop.Excel.Worksheet(book.Worksheets.Item(1));  
range = sheet.UsedRange;  
arr = range.Value;
```

Convert the data to MATLAB types.

```
data = cell(arr, 'ConvertTypes', {'all'});
```

Display the dates.

```
cellfun(@disp, data(:,1))
```

```
Date  
31-Oct-1996 00:00:00  
  
29-Nov-1996 00:00:00  
  
30-Dec-1996 00:00:00  
  
31-Jan-1997 00:00:00
```

Quit the Excel program.

`Close(book)`

`Quit(app)`

See Also

Related Examples

- “Read Spreadsheet Data Using Excel as Automation Server” on page 16-6

More About

- “Convert .NET Arrays to Cell Arrays” on page 15-66
- “Spreadsheets”

Access a Simple .NET Class

In this section...

“System.DateTime Example” on page 15-6
“Create .NET Object from Constructor” on page 15-7
“View Information About .NET Object” on page 15-7
“Introduction to .NET Data Types” on page 15-9

System.DateTime Example

This example shows how to access functionality already loaded on your system. The topics following the example introduce some key steps and ideas to help you get started using .NET in MATLAB.

The Microsoft .NET Framework class library contains classes, such as `System.DateTime`, you can use in MATLAB. The following code creates an object and uses `DateTime` properties and methods to display information about the current date and time.

```
% Create object for current date and time
netDate = System.DateTime.Now;

% Display properties
netDate.DayOfWeek
netDate.Hour

% Call methods
ToShortTimeString(netDate)
AddDays(netDate,7);

% Call static method
System.DateTime.DaysInMonth(netDate.Year,netDate.Month)
```

The following topics provide more information about creating and viewing information about objects and an introduction to .NET data types.

For information about the .NET Framework class library, refer to the third-party documentation described in “To Learn More About the .NET Framework” on page 15-21.

Create .NET Object from Constructor

The example in the previous section uses the `Now` property to create a `DateTime` object. The following example shows how to create an object using one of the `DateTime` constructors.

```
myDate = System.DateTime(2000,1,31);
```

To call this constructor, or any method, you need to know its argument list, or *function signature*. Your vendor product documentation shows the function signatures. You can also display the signatures using the MATLAB `methodsview` function. Type `methodsview('System.DateTime')` and search the list for `DateTime` entries, such as shown in the following table.

Return Type	Name	Arguments
System.DateTime obj	DateTime	(int32 scalar year,...)

From the .NET Class Framework documentation, the following signature initializes a new instance of the `DateTime` structure to the specified year, month, and day, which is the information required for the `myDate` variable.

Return Type	Name	Arguments
System.DateTime obj	DateTime	(int32 scalar year, int32 scalar month, int32 scalar day)

For more information, see “Reading Method Signatures” on page 15-87.

View Information About .NET Object

Although the vendor documentation contains information about `DateTime` objects, you can use MATLAB commands, like `properties` and `methods`, to display information about .NET objects. For example:

```
% Display an object
netDate = System.DateTime.Now
% Display its properties
properties System.DateTime
% Display its methods
methods System.DateTime
```

MATLAB displays the following information. (The property values reflect your specific date and time.)

Display of DateTime Object

```
netDate =  
System.DateTime  
Package: System  
  
Properties:  
    Date: [1x1 System.DateTime]  
    Day: 11  
    DayOfWeek: [1x1 System.DayOfWeek]  
    DayOfYear: 11  
    Hour: 12  
    Kind: [1x1 System.DateTimeKind]  
    Millisecond: 413  
    Minute: 31  
    Month: 1  
    Now: [1x1 System.DateTime]  
    UtcNow: [1x1 System.DateTime]  
    Second: 38  
    Ticks: 634303458984133595  
    TimeOfDay: [1x1 System.TimeSpan]  
    Today: [1x1 System.DateTime]  
    Year: 2011  
    MinValue: [1x1 System.DateTime]  
    MaxValue: [1x1 System.DateTime]  
Methods, Superclasses
```

Display of DateTime Properties

Properties for class System.DateTime:

```
Date  
Day  
DayOfWeek  
DayOfYear  
Hour  
Kind  
Millisecond  
Minute  
Month  
Now  
UtcNow
```



```

Second
Ticks
TimeOfDay
Today
Year
MinValue
MaxValue

```

Display of DateTime Methods

Methods for class System.DateTime:

Add	GetType	ToUniversalTime
AddDays	GetTypeCode	addListener
AddHours	IsDaylightSavingTime	delete
AddMilliseconds	Subtract	eq
AddMinutes	ToBinary	findobj
AddMonths	ToFileTime	findprop
AddSeconds	ToFileTimeUtc	ge
AddTicks	ToLocalTime	gt
AddYears	ToLongDateString	invalid
CompareTo	ToLongTimeString	le
DateTime	ToOADate	lt
Equals	ToShortDateString	ne
GetDateTimeFormats	ToShortTimeString	notify
GetHashCode	ToString	

Static methods:

Compare	Parse	op_GreaterThan
DaysInMonth	ParseExact	op_GreaterThanOrEqual
FromBinary	SpecifyKind	op_Inequality
FromFileTime	TryParse	op_LessThan
FromFileTimeUtc	TryParseExact	op_LessThanOrEqual
FromOADate	op_Addition	op_Subtraction
IsLeapYear	op_Equality	

For more information, see:

- “Using .NET Properties” on page 15-72
- “Calling .NET Methods” on page 15-86

Introduction to .NET Data Types

To use .NET objects in MATLAB, you need to understand how MATLAB treats .NET data types. For example, the following `DateTime` properties and methods create variables of various .NET types:

```

netDate = System.DateTime.Now;
thisDay = netDate.DayOfWeek;

```

```
thisHour = netDate.Hour;  
thisDate = ToLongDateString(netDate);  
thisTime = ToShortTimeString(netDate);  
monthSz = System.DateTime.DaysInMonth(netDate.Year, netDate.Month);  
whos
```

Name	Size	Bytes	Class
netDate	1x1	112	System.DateTime
monthSz	1x1	4	int32
thisDate	1x1	112	System.String
thisDay	1x1	104	System.DayOfWeek
thisHour	1x1	4	int32
thisTime	1x1	112	System.String

MATLAB displays the type as a class name.

To use these variables in MATLAB, consider the following:

- Numeric values (`int32`) — MATLAB preserves .NET numeric types by mapping them into equivalent MATLAB types. In the following example, `h` is type `int32`.

```
h = thisHour + 1;
```

For more information, see “.NET Type to MATLAB Type Mapping” on page 15-57 and “Numeric Types”.

- Strings (`System.String`) — Use the `char` function to convert a `System.String` object to a MATLAB char array:

```
disp(['The time is ' char(thisTime)])
```

- Objects (`System.DateTime`) — Refer to the .NET Framework class library documentation for information about using a `DateTime` object.
- Enumerations (`System.DayOfWeek`) — According to the `DateTime` documentation, `DayOfWeek` is an enumeration. To display the enumeration members, type:

```
enumeration(thisDay)
```

For more information, see “.NET Enumerations in MATLAB”.

For a complete list of supported types and mappings, see “Handle Data Returned from .NET Objects” on page 15-57.

See Also

More About

- “To Learn More About the .NET Framework” on page 15-21
- “Reading Method Signatures” on page 15-87
- “.NET Type to MATLAB Type Mapping” on page 15-57
- “Handle Data Returned from .NET Objects” on page 15-57

Load a Global .NET Assembly

This example shows you how to make .NET classes visible to MATLAB by loading a global assembly using the `NET.addAssembly` function.

The speech synthesizer class (available in .NET Framework Version 3.0 and above) provides ready-to-use text-to-speech features. For example, type:

```
NET.addAssembly('System.Speech');  
speak = System.Speech.Synthesis.SpeechSynthesizer;  
speak.Volume = 100;  
Speak(speak, 'You can use .NET Libraries in MATLAB')
```

The speech synthesizer class, like any .NET class, is part of an assembly. To work with the class, call `NET.addAssembly` to load the assembly into MATLAB. Your vendor documentation contains the assembly name. For example, search the Microsoft .NET Framework website for the `System.SpeechSynthesizer` class. The assembly name is `System.Speech`.

```
NET.addAssembly('System.Speech');
```

The `System.Speech` assembly is a global assembly. If your assembly is a private assembly, use the full path for the input to `NET.addAssembly`.

The “System.DateTime Example” on page 15-6 does not call `NET.addAssembly` because MATLAB dynamically loads its assembly (`microsoftlib`) at startup.

Note You cannot unload an assembly in MATLAB.

See Also

`NET.addAssembly`

More About

- “Assembly Is Library of .NET Classes” on page 15-16

Work with Microsoft Excel Spreadsheets Using .NET

This example creates a spreadsheet, copies some MATLAB data to it, and closes it. The example uses classes from the `Microsoft.Office.Interop.Excel.ApplicationClass` class. For information about the class and using the interface with different versions of Excel, refer to documentation on the MSDN website <https://msdn.microsoft.com/en-us/>.

To create a workbook, type:

```
NET.addAssembly('microsoft.office.interop.excel');
app = Microsoft.Office.Interop.Excel.ApplicationClass;
books = app.Workbooks;
newWB = Add(books);
app.Visible = true;
```

Create a sheet:

```
sheets = newWB.Worksheets;
newSheet = Item(sheets,1);
```

`newSheet` is a `System.__ComObject` because `sheets.Item` can return different types, such as a `Chart` or a `Worksheet`. To make the sheet a `Worksheet`, use the command:

```
newWS = Microsoft.Office.Interop.Excel.Worksheet(newSheet);
```

Create MATLAB data and write columns 1 and 2 to a range of cells.

```
excelArray = rand(10);
newRange = Range(newWS, 'A1');
newRange.Value2 = 'Data from Location A';
newRange = Range(newWS, 'A3:B12');
newRange.Value2 = excelArray;
```

Add three text strings to column C.

```
% Create a 3x1 System.Object
strArray = NET.createArray('System.Object',3,1);
strArray(1,1) = 'Add';
strArray(2,1) = 'text';
strArray(3,1) = 'to column C';
newRange = Range(newWS, 'C3:C5');
newRange.Value2 = strArray;
```

Modify cell format and name the worksheet:

```
newFont = newRange.Font;  
newFont.Bold = 1;  
newWS.Name = 'Test Data';
```

If this is a new spreadsheet, use the `SaveAs` method:

```
SaveAs(newWB, 'mySpreadsheet.xlsx');
```

Close and quit:

```
Close(newWB)  
Quit(app)
```

See Also

Related Examples

- “Work with Microsoft Word Documents Using .NET” on page 15-15

External Websites

- <https://msdn.microsoft.com/en-us/>

Work with Microsoft Word Documents Using .NET

This example uses classes from the `Microsoft.Office.Interop.Word.ApplicationClass` class. For information about the class and using the interface with different versions of Microsoft Word, refer to documentation on the MSDN website <https://msdn.microsoft.com/en-us/>.

The following code creates a new Word document:

```
NET.addAssembly('microsoft.office.interop.word');  
wordApp = Microsoft.Office.Interop.Word.ApplicationClass;  
wordDoc = wordApp.Documents;  
newDoc = Add(wordDoc);
```

If you want to type directly into the document, type the MATLAB command:

```
wordApp.Visible = true;
```

Put the cursor into the document window and enter text.

To name the document `myDocument.docx` and save it in the My Documents folder, type:

```
SaveAs(newDoc, 'myDocument.docx');
```

When you are finished, to close the document and application, type:

```
Save(newDoc);  
Close(newDoc);  
Quit(wordApp);
```

See Also

Related Examples

- “Work with Microsoft Excel Spreadsheets Using .NET” on page 15-13

External Websites

- <https://msdn.microsoft.com/en-us/>

Assembly Is Library of .NET Classes

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources built to work together and form a logical unit of functionality.

To work with a .NET application, you need to make its assemblies visible to MATLAB. How you do this depends on how the assembly is deployed, either privately or globally.

- global assembly—Shared among applications and installed in a common folder, called the Global Assembly Cache (GAC).
- private assembly—Used by a single application.

To load a global assembly into MATLAB, use the short name of the assembly, which is the file name without the extension. To load a private assembly, you need the full path (folder and file name with extension) of the assembly. This information is in the vendor documentation for the assembly. Refer to the vendor documentation for information about using your product.

The following assemblies from the .NET Framework class library are available at startup. MATLAB dynamically loads them the first time you type “NET.” or “System.”.

- `mscorlib.dll`
- `system.dll`

To use any other .NET assembly, load the assembly using the `NET.addAssembly` command. After loading the assembly, you can work with the classes defined by the assembly.

Note You cannot unload an assembly in MATLAB. If you modify and rebuild your own assembly, you must restart MATLAB to access the changes.

See Also

`NET.addAssembly`

Limitations to .NET Support

MATLAB supports the .NET features C# supports, except for the limits noted in the following table.

Features Not Supported in MATLAB
Cannot use <code>ClassName.propertyname</code> syntax to set static properties. Use <code>NET.setStaticProperty</code> instead.
Unloading an assembly
Passing a structure array, sparse array, or complex number to a .NET property or method
Subclassing .NET classes from MATLAB
Accessing nonpublic class members
Displaying generic methods using <code>methods</code> or <code>methodsview</code> functions. For a workaround, see “Display .NET Generic Methods Using Reflection” on page 15-135.
Creating an instance of a nested class. For a workaround, see “Nested Classes” on page 15-29.
Saving (serializing) .NET objects into a MAT-file
Creating .NET arrays with a specific lower bound
Concatenating multiple .NET objects into an array
Implementing interface methods
Hosting .NET controls in figure windows
Casting operations
Calling constructors with <code>ref</code> or <code>out</code> type arguments
Using <code>System.Console.WriteLine</code> to write text to the command window
Pointer type arguments, function pointers, <code>Dllimport</code> keyword
.NET remoting
Using the MATLAB <code>:</code> (colon) operator in a <code>foreach</code> iteration
Adding event listeners to .NET events defined in static classes
Handling .NET events with signatures that do not conform to the standard signature
Creating empty .NET objects
Creating .NET objects that do not belong to a namespace

See Also

More About

- “Limitations to Support of .NET Arrays” on page 15-69
- “Limitations to Support of .NET Methods” on page 15-93
- “Limitations to Support of .NET Events” on page 15-108
- “Limitations to Support of .NET Delegates” on page 15-109
- “Limitations to Support of .NET Enumerations” on page 15-130
- “System Requirements for Using MATLAB Interface to .NET” on page 15-19

System Requirements for Using MATLAB Interface to .NET

The MATLAB interface to .NET is available on the Windows platform only.

You must have the Microsoft .NET Framework installed on your system.

The MATLAB interface requires the .NET Framework Version 4.0 and above. The interface continues to support assemblies built on Framework 2.0 and above. To determine if your system has the supported framework, use the `NET.IsNETSupported` function.

To use a .NET application, refer to your vendors product documentation for information about how to install the program and for details about its functionality.

MATLAB Configuration File

MATLAB provides a configuration file, `MATLAB.exe.config`, in your *matlabroot/bin/64* folder. With this file, MATLAB loads the latest core assemblies available on your system. You can modify and use the configuration file at your own risk. For additional information on elements that can be used in the configuration file, visit the Configuration File Schema for the .NET Framework website at <https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/file-schema/index>.

See Also

More About

- “Limitations to .NET Support” on page 15-17

Using .NET from MATLAB

In this section...

“Benefits of the MATLAB .NET Interface” on page 15-20

“Why Use the MATLAB .NET Interface?” on page 15-20

“NET Assembly Integration Using MATLAB Compiler SDK” on page 15-21

“To Learn More About the .NET Framework” on page 15-21

Benefits of the MATLAB .NET Interface

The MATLAB .NET interface enables you to:

- Create instances of .NET classes.
- Interact with .NET applications via their class members.

Why Use the MATLAB .NET Interface?

Use the MATLAB .NET interface to take advantage of the capabilities of the Microsoft .NET Framework. For example:

- You have a professionally developed .NET assembly and want to use it to do certain operations, such as access hardware.
- You want to leverage the capabilities of programming in .NET (for example, you have existing C# programs).
- You want to access existing Microsoft-supplied classes for .NET.

The speech synthesizer class, available in .NET Framework Version 3.0 and above, is an example of a ready-to-use feature. Create the following `Speak` function in MATLAB:

```
function Speak(text)
NET.addAssembly('System.Speech');
speak = System.Speech.Synthesis.SpeechSynthesizer;
speak.Volume = 100;
Speak(speak, text)
end
```

For an example rendering text to speech, type:

Speak('You can use .NET Libraries in MATLAB')

NET Assembly Integration Using MATLAB Compiler SDK

The MATLAB .NET interface is for MATLAB users who want to use .NET assemblies in MATLAB.

NET Assembly Integration in the MATLAB Compiler SDK product packages MATLAB functions so that .NET programmers can access them. It brings MATLAB into .NET applications. For information about NET Assembly Integration, see the MATLAB Compiler SDK product documentation.

To Learn More About the .NET Framework

For a complete description of the .NET Framework, you need to consult outside resources.

One source of information is the Microsoft Developer Network. Search the .NET Framework Development Center for the term “.NET Framework Class Library”. The .NET Framework Class Library is a programming reference manual. Many examples in this documentation refer to classes in this library. There are different versions of the .NET Framework documentation, so be sure to refer to the version that is on your system. See “System Requirements for Using MATLAB Interface to .NET” on page 15-19 for information about version support in MATLAB.

See Also

More About

- “Limitations to .NET Support” on page 15-17

Using a .NET Object

In this section...

“Creating a .NET Object” on page 15-22

“What Classes Are in a .NET Assembly?” on page 15-22

“Using the delete Function on a .NET Object” on page 15-23

Creating a .NET Object

You often create objects when working with .NET classes. An object is an instance of a particular class. Methods are functions that operate exclusively on objects of a class. Data types package together objects and methods so that the methods operate on objects of their own type. For information about using objects in MATLAB, see “Operations on Objects”.

You construct .NET objects in the MATLAB workspace by calling the class constructor, which has the same name as the class. The syntax to create a .NET object `classObj` is:

```
classObj = namespace.ClassName(varargin)
```

where `varargin` is the list of constructor arguments to create an instance of the class specified by `ClassName` in the given namespace. For an example, see “Create .NET Object from Constructor” on page 15-7.

To call method `methodName`:

```
returnedValue = methodName(classObj, args, ...)
```

What Classes Are in a .NET Assembly?

The product documentation for your assembly contains information about its classes. However, you can use the `NET.addAssembly` command to read basic information about an assembly.

For example, to view the class names of the `microsoftlib` library, type:

```
asm = NET.addAssembly('mscorlib');  
asm.Classes
```

This assembly has hundreds of entries. You can open a window to the online document for the `System` namespace reference page on the Microsoft Developer Network. For information about using this documentation, see “To Learn More About the .NET Framework” on page 15-21.

Using the delete Function on a .NET Object

Objects created from .NET classes appear in MATLAB as reference types, or handle objects. Calling the `delete` function on a .NET handle releases all references to that .NET object from MATLAB, but does not invoke any .NET finalizers. The .NET Framework manages garbage collection.

For more information about managing handle objects in MATLAB, see “Handle Class Destructor”.

See Also

More About

- “.NET Properties in MATLAB”
- “.NET Methods in MATLAB”
- “.NET Events and Delegates in MATLAB”
- “Role of Classes in MATLAB”
- “Operations on Objects”

Build a .NET Application for MATLAB Examples

You can use C# code examples in MATLAB, such as the `NetDocCell` assembly provided in “Convert .NET Arrays to Cell Arrays” on page 15-66. Build an application using a C# development tool, like Microsoft Visual Studio and then load it into MATLAB using the `NET.addAssembly` function. The following are basic steps for building; consult your development tool documentation for specific instructions.

- 1 From your development tool, open a new project and create a C# class library.
- 2 Copy the classes and other constructs from the C# files into your project.
- 3 Build the project as a DLL.
- 4 The name of this assembly is the namespace. Note the full path to the DLL file. Since it is a private assembly, you must use the full path to load it in MATLAB.
- 5 After you load the assembly, if you modify and rebuild it, you must restart MATLAB to access the new assembly. You cannot unload an assembly in MATLAB.

See Also

More About

- “Access a Simple .NET Class” on page 15-6
- “Using .NET Properties” on page 15-72
- “Pass Cell Arrays of .NET Data” on page 15-40
- “Work with Members of a .NET Enumeration” on page 15-119
- “Call .NET Generic Methods” on page 15-140

Troubleshooting Security Policy Settings from Network Drives

If you run a .NET command on a MATLAB session started from a network drive, you could see a warning message. To resolve this problem, run the `enableNETfromNetworkDrive` function.

This file adds the following entry to the security policy on your machine to trust the `dotnetcli` assembly, which is the MATLAB interface to .NET module:

- Creates a group named `MathWorks_Zone` with `LocalIntranet` permission.
- Creates a `dotnetcli` subgroup within `MathWorks_Zone`.
- Provides `Full-Trust` to the `dotnetcli.dll` strong name for access to the local intranet.

You must have administrative privileges to change your configuration.

.NET Terminology

A namespace is a way to group identifiers. A namespace can contain other namespaces. In MATLAB, a namespace is a package. In MATLAB, a .NET type is a class.

The syntax `namespace.ClassName` is known as a fully qualified name.

.NET Framework System Namespace

`System` is the root namespace for fundamental types in the .NET Framework. This namespace also contains classes (for example, `System.String` and `System.Array`) and second-level namespaces (for example, `System.Collections.Generic`). The `mscorlib` and `system` assemblies, which MATLAB loads at startup, contain many, but not all `System` namespaces. For example, to use classes in the `System.Xml` namespace, load the `system.xml` assembly using the `NET.addAssembly` command. Refer to the Microsoft .NET Framework Class Library Reference to learn what assembly to use for a specific namespace.

Reference Type Versus Value Type

Objects created from .NET classes (for example, the `System.Reflection.Assembly` class) appear in MATLAB as reference types, or handle objects. Objects created from .NET structures (for example, the `System.DateTime` structure) appear as value types. You use the same MATLAB syntax to create and access members of classes and structures.

However, handle objects are different from value objects. When you copy a handle object, only the handle is copied and both the old and new handles refer to the same data. When you copy a value object, the object data is also copied and the new object is independent of changes to the original object. For more information about these differences, see “Object Behavior”.

Do not confuse an object created from a .NET structure with a MATLAB structure array (see “Structures”). You cannot pass a structure array to a .NET method.

Simplify .NET Class Names

In a MATLAB command, you can refer to any class by its fully qualified name, which includes its package name. A fully qualified name might be long, making commands and functions, such as constructors, cumbersome to edit and to read. You can refer to classes by the class name alone (without a package name) if you first import the fully qualified name into MATLAB. The `import` function adds all classes that you import to a list called the import list. You can see what classes are on that list by typing `import`, without any arguments.

For example, to eliminate the need to type `System.` before every command in the “Access a Simple .NET Class” on page 15-6 example, type:

```
import System.*  
import System.DateTime.*
```

To create the object, type:

```
netDate = DateTime.Today;
```

To use a static method, type:

```
DaysInMonth(netDate.Year, netDate.Month)
```

See Also

`import`

Use import in MATLAB Functions

If you use the `import` command in a MATLAB function, add the corresponding .NET assembly before calling the function. For example, the following function `getPrinterInfo` calls methods in the `System.Drawing` namespace.

```
function ptr = getPrinterInfo
import System.Drawing.Printing.*;
ptr = PrinterSettings;
end
```

To call the function, type:

```
NET.addAssembly('System.Drawing');
printer = getPrinterInfo;
```

Do not add the command `NET.addAssembly('System.Drawing')` to the `getPrinterInfo` function. MATLAB processes the `getPrinterInfo.m` code before executing the `NET.addAssembly` command. In that case, `PrinterSettings` is not fully qualified and MATLAB does not recognize the name.

Likewise, the scope of the `import` command is limited to the `getPrinterInfo` function. At the command line, type:

```
ptr = PrinterSettings;
```

```
Undefined function or variable 'PrinterSettings'.
```

See Also

`import`

Nested Classes

In MATLAB, you cannot directly instantiate a nested class but here is how to do it through reflection. The following C# code defines `InnerClass` nested in `OuterClass`:

```
namespace MyClassLibrary
{
    public class OuterClass
    {
        public class InnerClass
        {
            public String strmethod(String x)
            {
                return "from InnerClass " + x;
            }
        }
    }
}
```

If the `MyClassLibrary` assembly is in your `c:\work` folder, load the file:

```
a = NET.addAssembly('C:\Work\MyClassLibrary.dll');
a.Classes
```

```
ans =
    'MyClassLibrary.OuterClass'
    'MyClassLibrary.OuterClass+InnerClass'
```

To call `strmethod`, type:

```
t = a.AssemblyHandle.GetType('MyClassLibrary.OuterClass+InnerClass');
sa = System.Activator.CreateInstance(t);
strmethod(sa, 'hello')
```

```
ans =
from InnerClass hello
```

Handle .NET Exceptions

MATLAB catches exceptions thrown by .NET and converts them into a `NET.NetException` object, which is derived from the `MException` class. The default display of `NetException` contains the `Message`, `Source` and `HelpLink` fields of the `System.Exception` class that caused the exception. For example:

```
try
    NET.addAssembly('C:\Work\invalidfile.dll')
catch e
    e.message
    if(isa(e,'NET.NetException'))
        e.ExceptionObject
    end
end
```

See Also

`NET.NetException`

Pass Numeric Arguments

In this section...
“Call .NET Methods with Numeric Arguments” on page 15-31
“Use .NET Numeric Types in MATLAB” on page 15-31

Call .NET Methods with Numeric Arguments

When you call a .NET method in MATLAB, MATLAB automatically converts numeric arguments into equivalent .NET types, as shown in the table in “Pass Primitive .NET Types” on page 15-49.

Use .NET Numeric Types in MATLAB

MATLAB automatically converts numeric data returned from a .NET method into equivalent MATLAB types, as shown in the table in “.NET Type to MATLAB Type Mapping” on page 15-57.

Note that MATLAB preserves .NET arrays as the relevant `System.Array` types, for example, `System.Double[]`.

MATLAB has rules for handling integers. If you are familiar with using integer types in MATLAB, and just need a reference to the rules, see the links at the end of this topic.

The default data type in MATLAB is `double`. If the data in your applications uses the default, then you need to pay attention to the numeric outputs of your .NET applications.

For more information, see:

- “Numeric Types”
- “Valid Combinations of Unlike Classes”
- “Combining Unlike Integer Types”
- “Integers”

Pass System.String Arguments

In this section...

“Call .NET Methods with System.String Arguments” on page 15-32

“Use System.String in MATLAB” on page 15-32

Call .NET Methods with System.String Arguments

If an input argument to a .NET method is `System.String`, you can pass a MATLAB string scalar or character array. MATLAB automatically converts the argument into `System.String`. For example, the following code uses the `System.DateTime.Parse` method to convert a date represented by a char array into a `DateTime` object:

```
strDate = "01 Jul 2010 3:33:02 GMT";
convertedDate = System.DateTime.Parse(strDate);
ToShortTimeString(convertedDate)
ToLongDateString(convertedDate)
```

To view the function signature for the `System.DateTime.Parse` method, type:

```
methodsview("System.DateTime")
```

Search the list for `Parse`.

Qualifiers	Return Type	Name	Arguments
Static	System.DateTime RetVal	Parse	(System.String s)

For more information, see:

- “Pass MATLAB String and Character Data” on page 15-51
- Search the MSDN website at <https://msdn.microsoft.com/en-us/> for the term `System.DateTime`.

Use System.String in MATLAB

This example shows how to use a `System.String` object in a MATLAB® function.

Create an object representing the current time. The current time `thisTime` is a `System.String` object.

```
netDate = System.DateTime.Now;  
thisTime = ToShortTimeString(netDate);  
class(thisTime)
```

```
ans =
```

```
    'System.String'
```

To display `thisTime` in MATLAB, use the `string` function to convert the `System.String` object to a MATLAB string.

```
join(["The time is", string(thisTime)])
```

```
ans =
```

```
    "The time is 13:21"
```

See Also

More About

- “How MATLAB Handles `System.String`” on page 15-59

Pass System.Enum Arguments

In this section...

“Call .NET Methods with System.Enum Arguments” on page 15-34

“Use System.Enum in MATLAB” on page 15-35

Call .NET Methods with System.Enum Arguments

An example of an enumeration is `System.DayOfWeek`. To see how to call a .NET method with this input type, use the `GetAbbreviatedDayName` method in the `System.Globalization.DateTimeFormatInfo` class. The following code displays the abbreviation for “Thursday”.

```
% Create a DayOfWeek object
thisDay = System.DayOfWeek.Thursday;
dtformat = System.Globalization.DateTimeFormatInfo;
% Display the abbreviated name of the specified day based on the
% culture associated with the current DateTimeFormatInfo object.
dtformat.GetAbbreviatedDayName(thisDay)
```

To view the function signature for the `GetAbbreviatedDayName` method, type:

```
methodsview('System.Globalization.DateTimeFormatInfo')
```

Search the list for `GetAbbreviatedDayName`.

Return Type	Name	Arguments
System.String RetVal	GetAbbreviatedDayName	(System.Globalization.DateTimeFormatInfo this, System.DayOfWeek dayofweek)

For more information, search the MSDN website at <https://msdn.microsoft.com/en-us/> for the term `DateTimeFormatInfo`.

Use System.Enum in MATLAB

In MATLAB, an enumeration is a class having a finite set of named instances. You can work with .NET enumerations using features of the MATLAB enumeration class and some features unique to the .NET Framework. Some ways to use the `System.DayOfWeek` enumeration in MATLAB:

- Display an enumeration member. For example:

```
myDay = System.DateTime.Today;
disp(myDay.DayOfWeek)
```

- Use an enumeration in comparison statements. For example:

```
myDay = System.DateTime.Today;
switch(myDay.DayOfWeek)
    case {System.DayOfWeek.Saturday, System.DayOfWeek.Sunday}
        disp('Weekend')
    otherwise
        disp('Work day')
end
```

- Perform calculations. For example, the underlying type of `DayOfWeek` is `System.Int32` which you can use to perform integer arithmetic. To display the date of the first day of the current week, type:

```
myDay = System.DateTime.Today;
dow = myDay.DayOfWeek;
startDateOfWeek = AddDays(myDay, -double(dow));
ToShortDateString(startDateOfWeek)
```

- Perform bitwise operations. For examples, see “Creating .NET Enumeration Bit Flags” on page 15-110.

For more information, see:

- “Iterate Through a .NET Enumeration” on page 15-125
- “Use .NET Enumerations to Test for Conditions” on page 15-127
- “Use Bit Flags with .NET Enumerations” on page 15-110

Pass System.Nullable Arguments

This example shows how to handle .NET methods with `System.Nullable` type arguments, whose underlying value type is `double`.

The example shows how to call a method with a `System.Nullable` input argument. It uses the MATLAB `plot` function to show to handle a `System.Nullable` output argument.

Build Custom Assembly NetDocNullable

To execute the MATLAB code in this example, build the `NetDocNullable` assembly. The assembly is created with the C# code, `NetDocNullable.cs`, in the `matlabroot/extern/examples/NET/NetSample` folder. To see the code, open the file in MATLAB Editor and build the `NetDocNullable` assembly.

`NetDocNullable` defines method `SetField` which has `System.Nullable` arguments.

SetField Function Signature

Return Type	Name	Arguments
<code>System.Nullable<System*Double></code> <code>RetVal</code>	<code>SetField</code>	<code>(NetDocNullable.MyClass this, System.Nullable<System*Double> db)</code>

Load NetDocNullable Assembly

The example assumes that you put the assembly in your `c:\work` folder. You can modify the example to change the path, `dllPath`, of the assembly.

```
dllPath = fullfile('c:', 'work', 'NetDocNullable.dll');
asm = NET.addAssembly(dllPath);
cls = NetDocNullable.MyClass;
```

Use the `cls` variable to call `SetField`, which creates a `System.Nullable<System*Double>` value from your input.

Pass System.Nullable Input Arguments

MATLAB automatically converts `double` and `null` values to `System.Nullable<System*Double>` objects.

Pass a double value.

```
field1 = SetField(cls,10)

field1 =
  System.Nullable<System*Double>
  Package: System

  Properties:
    HasValue: 1
    Value: 10
  Methods, Superclasses
```

The `HasValue` property is true (1) and the `Value` property is 10.

Pass null value, [].

```
field2 = SetField(cls,[])

field2 =
  System.Nullable<System*Double>
  Package: System

  Properties:
    HasValue: 0
  Methods, Superclasses
```

The `HasValue` property is false (0), and it has no `Value` property.

Handle System.Nullable Output Arguments in MATLAB

Before you use a `System.Nullable` object in MATLAB, first decide how to handle null values. If you ignore null values, you might get unexpected results when you use the value in a MATLAB function.

The `System.Nullable` class provides two techniques for handling null values. To provide special handling for null values, use the `HasValue` property. To treat a null value in the same way as a double, use the `GetValueOrDefault` method.

Create a MATLAB function, `plotValue.m`, which detects null values and treats them differently from numeric values. The input is a `System.Nullable<System*Double>` type. If the input is null, the function displays a message. If the input value is double, it creates a line graph from 0 to the value.

```
function plotValue(x)
% x is System.Nullable<System*Double> type
if (x.HasValue && isfloat(x.Value))
    plot([0 x.Value])
else
    disp('No Data')
end
```

The `plotValue` function uses the `HasValue` property of the input argument to detect null values and calls the MATLAB `plot` function using the `Value` property.

Call `plotValue` with variable `field1` to display a line graph.

```
plotValue(field1)
```

Call `plotValue` with the variable `field2`, a null value.

```
plotValue(field2)
```

```
No Data
```

If you do not need special processing for null values, use the `GetValueOrDefault` method. To display the `GetValueOrDefault` function signature, type:

```
methodsview(field1)
```

Look for the following function signature:

GetValueOrDefault Function Signature

Return Type	Name	Arguments
double scalar RetVal	GetValueOrDefault	(System.Nullable <System*Double> this)

This method converts the input variable to `double` so you can directly call the MATLAB `plot` function:

```
myData = GetValueOrDefault(field1);
plot([0 myData+2])
```

The `GetValueOrDefault` method converts a null value to the default numeric value, `0`.

```
defaultData = GetValueOrDefault(field2)
```

```
defaultData =
    0
```

Call `plot`:

```
plot([0 defaultData])
```

You can change the default value using the `GetValueOrDefault` method. Open the `methodsvew` window and look for the following function signature:

GetValueOrDefault Function Signature to Change Default

Return Type	Name	Arguments
double scalar RetVal	GetValueOrDefault	(System.Nullable<System*Double> this, double scalar defaultValue)

Set the `defaultValue` input argument to a new value, `-1`, and plot the results for null value `field2`.

```
defaultData = GetValueOrDefault(field2, -1);
plot([0 defaultData])
```

See Also

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24

More About

- “Pass System.Nullable Type” on page 15-52
- “How MATLAB Handles System.Nullable” on page 15-61

External Websites

- <https://msdn.microsoft.com/en-us/>

Pass Cell Arrays of .NET Data

In this section...

“Example of Cell Arrays of .NET Data” on page 15-40

“Create a Cell Array for Each System.Object” on page 15-41

“Create MATLAB Variables from the .NET Data” on page 15-41

“Call MATLAB Functions with MATLAB Variables” on page 15-41

Example of Cell Arrays of .NET Data

In the “Convert Nested System.Object Arrays” on page 15-66 example, the cell array `mldata` contains data from the `MyGraph.getNewData` method. By reading the class documentation in the source file, you can create the following MATLAB graph:

```
dllPath = fullfile('c:', 'work', 'NetDocCell.dll');
asm = NET.addAssembly(dllPath);
graph = NetDocCell.MyGraph;
```

```
% Create cell array containing all data
mldata = cell(graph.getNewData);
```

```
% Plot the data and label the graph
figure('Name', char(mldata{1}))
plot(double(mldata{2}(2)))
xlabel(char(mldata{2}(1)))
```

However, keeping track of data of different types and dimensions and the conversions necessary to map .NET data into MATLAB types is complicated using the cell array structure. Here are some tips for working with the contents of nested `System.Object` arrays in MATLAB. After reading data from a .NET method:

- Create cell arrays for all `System.Object` arrays.
- Convert the .NET types to MATLAB types, according to the information in “Handle Data Returned from .NET Objects” on page 15-57.
- Create MATLAB variables for each type within the cell arrays.
- Call MATLAB functions with the MATLAB variables.

Create a Cell Array for Each System.Object

This example shows how to copy `System.Object` data into a cell array.

The following statement creates the cell array `mlData`:

```
mlData = cell(graph.getNewData)

mlData =
    [1x1 System.String]    [1x1 System.Object[]]
```

This cell array contains elements of these types.

To access the contents of the `System.Object` array, create another cell array `mlPlotData`:

```
mlPlotData = cell(mlData{2})

mlPlotData =
    [1x1 System.String]    [1x1 System.Double[]]
```

This cell array contains elements of these types.

Create MATLAB Variables from the .NET Data

Assign cell data to MATLAB variables and convert:

```
% Create descriptive variables
% Convert System.String to char
mytitle = char(mlData{1});
myxlabel = char(mlPlotData{1});
% Convert System.Double to double
y = double(mlPlotData{2});
```

Call MATLAB Functions with MATLAB Variables

Create a MATLAB graph with this data:

```
% Remove the previous figure
close
% Plot the data and label the graph
figure('Name',mytitle,'NumberTitle','off')
```

```
plot(y)  
xlabel(myxlabel)
```

Pass Jagged Arrays

In this section...

“Create System.Double .NET Jagged Array” on page 15-43

“Call .NET Method with System.String Jagged Array Arguments” on page 15-43

“Call .NET Method with Multidimensional Jagged Array Arguments” on page 15-45

Create System.Double .NET Jagged Array

This example shows how to create a .NET jagged array of System.Double using the NET.createArray function.

Create a three element array. You can pass jArr to any .NET method with an input or output argument of type System.Double[][].

```
jArr = NET.createArray('System.Double[]',3)
```

```
jArr =
```

```
Double[][] with properties:
```

```

    Length: 3
  LongLength: 3
    Rank: 1
  SyncRoot: [1x1 System.Double[][]]
  IsReadOnly: 0
  IsFixedSize: 1
  IsSynchronized: 0

```

Call .NET Method with System.String Jagged Array Arguments

This example shows how to create an array of MATLAB character vectors to pass to a method, MethodStringArr, with a System.String[][] input argument.

The following is the MATLAB function signature for MethodStringArr.

Return Type	Name	Arguments
System.String[][] RetVal	MethodStringArr	(NetPackage.StringClass this, System.String[][] arr)

The MATLAB character vectors you want to pass to the method are:

```
str1 = {'this', 'is'};
str2 = 'jagged';
```

Create a variable, `netArr`, of `System.String` arrays, which contains two arrays. Using the `NET.createArray`, the `typeName` for this array is `System.String[]`, and the dimension is 2.

```
netArr = NET.createArray('System.String[]',2);
```

The arrays contain empty strings.

Create `System.String` arrays to correspond to the MATLAB character vectors, `str1` and `str2`.

```
netArr(1) = NET.createArray('System.String',2);
netArr(2) = NET.createArray('System.String',1);
```

Assign `str1` and `str2` to `netArr`.

```
netArr(1) = str1;
netArr(2,1) = str2;
```

Because `str2` is a scalar and `netArr(2)` expects an array, you must assign `str2` to the specific element `netArr(2,1)`.

Now you can pass `netArr` to the `MethodStringArr` method.

```
class(netArr)
```

```
ans =
System.String[][]
```

Call .NET Method with Multidimensional Jagged Array Arguments

This example shows how to create a MATLAB array to pass to a method, `MethodMultiDarr`, with a multidimensional jagged array input argument of `System.Double` type.

The following is the MATLAB function signature for `MethodMultiDarr`. The input is a multidimensional jagged array that contains single dimensional elements.

Return Type	Name	Arguments
System.Double[][][,] RetVal	MethodMultiDarr	(NetPackage.NumericClass this, System.Double[][][,] arr)

Create a 2-by-3 array with `typeName` of `System.Double[]`.

```
arr = NET.createArray('System.Double[]',2,3);
```

The elements are empty arrays.

The MATLAB arrays you want to pass to the method are:

```
A1 = [1 2 3];
A2 = [5 6 7 8];
```

MATLAB automatically converts a numeric array to the equivalent .NET type.

```
arr(1,1) = A1;
arr(1,2) = A2;
```

Array `arr` is a `System.Double[][][,]` jagged array.

```
arr
```

```
arr =
```

```
Double[][][,] with properties:
```

```
Length: 6
LongLength: 6
Rank: 2
SyncRoot: [1x1 System.Double[,] ]
IsReadOnly: 0
IsFixedSize: 1
IsSynchronized: 0
```

Now you can pass `arr` to the `MethodMultiDArr` method.

Convert Nested System.Object Arrays

This example shows how to use the `cell` function to convert data in nested `System.Object` arrays.

The conversion of .NET arrays to cell arrays is not recursive for a `System.Object` array contained within a `System.Object` array. Use the `cell` function to convert each `System.Object` array.

The C# example `NetDocCell.cs`, in the `matlabroot/extern/examples/NET/NetSample` folder, is used in the following example. To see the code, open the file in MATLAB Editor and build the `NetDocCell` assembly.

Set up the path to your assembly, then load the assembly.

```
dllPath = fullfile('c:', 'work', 'NetDocCell.dll');  
NET.addAssembly(dllPath);
```

Create a cell array, `mldata`.

```
graph = NetDocCell.MyGraph;  
mldata = cell(graph.getNewData)
```

```
mldata =  
    [1x1 System.String]    [1x1 System.Object[]]
```

To access the contents of the `System.Object` array, create another cell array `mldata`.

```
mldata = cell(mldata{2})
```

```
mldata =  
    [1x1 System.String]    [1x1 System.Double[]]
```

See Also

Related Examples

- “Pass Cell Arrays of .NET Data” on page 15-40

More About

- “Build a .NET Application for MATLAB Examples” on page 15-24

Pass Data to .NET Objects

When you call a .NET method or function from MATLAB, MATLAB automatically converts arguments into .NET types. MATLAB performs this conversion on each passed argument, except for arguments that are already .NET objects. The following topics provide information about passing specific data types to .NET methods.

Pass Primitive .NET Types

The following table shows the MATLAB base types for passed arguments and the corresponding .NET types defined for input arguments. Each row shows a MATLAB type followed by the possible .NET argument matches, from left to right in order of closeness of the match.

MATLAB Primitive Type Conversion Table

MATLAB Type	Closest Type <----- Other Matching .NET Types -----> Least Close Type Preface Each .NET Type with System.											
logical	Boolean	Byte	SByte	Int16	UInt16	Int32	UInt32	Int64	UInt64	Single	Double	Object
double	Double	Single	Decimal	Int64	UInt64	Int32	UInt32	Int16	UInt16	SByte	Byte	Object
single	Single	Double	Decimal	Object								
int8	SByte	Int16	Int32	Int64	Single	Double	Object					
uint8	Byte	UInt16	UInt32	UInt64	Single	Double	Object					
int16	Int16	Int32	Int64	Single	Double	Object						
uint16	UInt16	UInt32	UInt64	Single	Double	Object						
int32	Int32	Int64	Single	Double	Object							
uint32	UInt32	UInt64	Single	Double	Object							
int64	Int64	Double	Object									
uint64	UInt64	Double	Object									
char	Char	String	Object									
string	String	Object										

The following primitive .NET argument types do not have direct MATLAB equivalent types. MATLAB passes these types as is:

- System.IntPtr
- System.UIntPtr
- System.Decimal
- enumerated types

Pass Cell Arrays

You can pass a cell array to a .NET property or method expecting an array of System.Object or System.String arguments, as shown in the following table.

MATLAB Cell Array Conversion Table

MATLAB Type	Closest Type <----- Other Matching .NET Types -----> Least Close Type		
Cell array of string scalars and/or character arrays	System.String[]	System.Object[]	System.Object
Cell array (no string or character arrays)	System.Object[]	System.Object	

Elements of a cell can be any of the following supported types:

- Any non-sparse, non-complex built-in numeric type shown in the MATLAB Primitive Type Conversion Table
- string
- char
- logical
- cell array
- .NET object

Pass Nonprimitive .NET Objects

When calling a method that has an argument of a particular .NET class, pass an object that is an instance of that class or its derived classes. You can create such an object using the class constructor, or use an object returned by a member of the class. When a class member returns a .NET object, MATLAB leaves it as a .NET object. Use this object to interact with other class members.

Pass MATLAB String and Character Data

MATLAB automatically converts:

- char array to a .NET System.String object. To pass an array of char arrays, create a cell array.
- string scalar to a .NET System.String object.
- Each string scalar in a string array to a .NET System.String object. The string array is converted to System.String[].

- String value <missing> to null.
- `string.empty` to `System.String[]` with size of 0.

Pass System.Nullable Type

You can pass any of the following to a .NET method with `System.Nullable<ValueType>` input arguments:

- Variable of the underlying `<ValueType>`
- null value, `[]`
- `System.Nullable<ValueType>` object

When you pass a MATLAB variable of type `ValueType`, MATLAB reads the signature and automatically converts your variable to a `System.Nullable<ValueType>` object. For a complete list of possible `ValueType` values accepted for `System.Nullable<ValueType>`, refer to the MATLAB Primitive Type Conversion Table.

For examples, see “Pass System.Nullable Arguments” on page 15-36.

Pass NULL Values

MATLAB uses empty double (`[]`) values for reference type arguments.

Unsupported MATLAB Types

MATLAB does not support passing the following MATLAB types to .NET methods:

- Structure arrays
- Sparse arrays
- Complex numbers

Choosing Method Signatures

MATLAB chooses the correct .NET method signature (including constructor, static and nonstatic methods) based on the following criteria.

When your MATLAB function calls a .NET method, MATLAB:

- 1 Checks to make sure that the object (or class, for a static method) has a method by that name.
- 2 Determines whether the invocation passes the same number of arguments of at least one method with that name.
- 3 Makes sure that each passed argument can be converted to the type defined for the method.

If all the preceding conditions are satisfied, MATLAB calls the method.

In a call to an overloaded method, if there is more than one candidate, MATLAB selects the one with arguments that best fit the calling arguments, based on the MATLAB Primitive Type Conversion Table. First, MATLAB rejects all methods that have any argument types that are incompatible with the passed arguments. Among the remaining methods, MATLAB selects the one with the highest fitness value, which is the sum of the fitness values of all its arguments. The fitness value for each argument is how close the MATLAB type is to the .NET type. If two methods have the same fitness, MATLAB chooses the first one defined in the class.

For class types, MATLAB chooses the method signature based on the distance of the incoming class type to the expected .NET class type. The closer the incoming type is to the expected type, the better the match.

The rules for overloaded methods with optional arguments are described in “Determining Which Overloaded Method Is Invoked” on page 15-88.

Example — Choosing a Method Signature

Open a `methodsvew` window for the `System.String` class and look at the entries for the `Concat` method:

```
import System.*
methodsvew('System.String')
```

The `Concat` method takes one or more arguments. If the arguments are of type `System.String`, the method concatenates the values. For example, create two strings:

```
str1 = String('hello');
str2 = String('world');
```

When you type:

```
String.Concat(str1,str2)
```

MATLAB verifies the method `Concat` exists and looks for a signature with two input arguments. The following table shows the two signatures.

Qualifiers	Return Type	Name	Arguments
Static	System.String RetVal	Concat	(System.Object arg0, System.Object arg1)
Static	System.String RetVal	Concat	(System.String str0, System.String str1)

Since `str1` and `str2` are of class `System.String`, MATLAB chooses the second signature and displays:

```
ans =  
helloworld
```

If the arguments are of type `System.Object`, the method displays the string representations of the values. For example, create two `System.DateTime` objects:

```
dt = DateTime.Today;  
myDate = System.DateTime(dt.Year,3,1,11,32,5);
```

When you type:

```
String.Concat(dt,myDate)
```

MATLAB chooses the following signature, since `System.DateTime` objects are derived from the `System.Object` class.

Qualifiers	Return Type	Name	Arguments
Static	System.String RetVal	Concat	(System.Object arg0, System.Object arg1)

This `Concat` method first applies the `ToString` method to the objects, then concatenates the strings. MATLAB displays information like:

```
ans =  
12/23/2008 12:00:00 AM3/1/2008 11:32:05 AM
```

Pass Arrays

For information about passing MATLAB arrays to .NET methods, see “Use Arrays with .NET Applications” on page 15-63 and “Pass MATLAB Arrays as Jagged Arrays” on page 15-55.

How Array Dimensions Affect Conversion

The dimension of a .NET array is the number of subscripts required to access an element of the array. To get the number of dimensions, use the `Rank` property of the `.NET System.Array` type. The dimensionality of a MATLAB array is the number of non-singleton dimensions in the array.

MATLAB matches the array dimensionality with the .NET method signature, as long as the dimensionality of the MATLAB array is lower than or equal to the expected dimensionality. For example, you can pass a scalar input to a method that expects a 2-D array.

For a MATLAB array with number of dimensions, *N*, if the .NET array has fewer than *N* dimensions, the MATLAB conversion drops singleton dimensions, starting with the first one, until the number of remaining dimensions matches the number of dimensions in the .NET array.

Converting a MATLAB Array to System.Object

You can pass a MATLAB array to a method that expects a `System.Object`.

Pass MATLAB Arrays as Jagged Arrays

A MATLAB array is a rectangular array. The .NET Framework supports a *jagged array*, which is an array of arrays. So the elements of a jagged array can be of different dimensions and sizes.

Although .NET languages support jagged arrays, the term *jagged* is not a language keyword. C# function signatures use multiple pairs of square brackets (`[][]`) to represent a jagged array. In addition, a jagged array can be nested (`[][][]`), multidimensional (`[,]`), or nested with multidimensional elements (for example, `[, ,][,] []`).

MATLAB automatically converts MATLAB arrays of numeric types to the corresponding jagged array type. If the input argument is a nonnumeric type or multidimensional, use

the `NET.createArray` function to create an array to pass as a jagged array. For examples using `NET.createArray`, see “Pass Jagged Arrays” on page 15-43.

Handle Data Returned from .NET Objects

In this section...

“.NET Type to MATLAB Type Mapping” on page 15-57
 “Convert Arrays of Primitive .NET Type to MATLAB Type” on page 15-58
 “How MATLAB Handles System.String” on page 15-59
 “How MATLAB Handles System.__ComObject” on page 15-59
 “How MATLAB Handles System.Nullable” on page 15-61
 “How MATLAB Handles dynamic Type” on page 15-61
 “How MATLAB Handles Jagged Arrays” on page 15-62

.NET Type to MATLAB Type Mapping

MATLAB automatically converts data from a .NET object into these MATLAB types. These values are displayed in a method signature.

C# .NET Type	MATLAB Type
System.Int16	int16 scalar
System.UInt16	uint16 scalar
System.Int32	int32 scalar
System.UInt32	uint32 scalar
System.Int64	int64 scalar
System.UInt64	uint64 scalar
System.Single	single scalar
System.Double	double scalar
System.Boolean	logical scalar
System.Byte	uint8 scalar
System.Enum	enum
System.Char	char
System.Decimal	System.Decimal
System.Object	System.Object

C# .NET Type	MATLAB Type
System.IntPtr	System.IntPtr
System.UIntPtr	System.UIntPtr
System.String	System.String
System.Nullable<ValueType>	System.Nullable<ValueType>
System.Array	See “Use Arrays with .NET Applications” on page 15-63
System.__ComObject	See “How MATLAB Handles System.__ComObject” on page 15-59
<i>class name</i>	<i>class name</i>
<i>struct name</i>	<i>struct name</i>

Convert Arrays of Primitive .NET Type to MATLAB Type

To convert elements of a .NET array to an equivalent MATLAB array, call these MATLAB functions. For an example, see “Convert Primitive .NET Arrays to MATLAB Arrays” on page 15-63.

C# .NET Type	MATLAB Conversion Function
System.Int16	int16
System.UInt16	uint16
System.Int32	int32
System.UInt32	uint32
System.Int64	int64
System.UInt64	uint64
System.Single	single
System.Double	double
System.Boolean	logical
System.Byte	uint8

How MATLAB Handles System.String

To convert a `System.String` object to a MATLAB string, use the `string` function. To convert a `System.String` object to a MATLAB character array, use the `char` function. For example:

```
str = System.String('create a System.String');
mlstr = string(str)
mlchar = char(str)

mlstr =

    "create a System.String"

mlchar =

    'create a System.String'
```

MATLAB displays the string value of `System.String` objects, instead of the standard object display. For example, type:

```
a = System.String('test')
b = System.String.Concat(a, ' hello', ' world')

a =
test
b =
test hello world
```

The `System.String` class illustrates how MATLAB handles fields and properties, as described in “Call .NET Properties That Take an Argument” on page 15-90. To see reference information about the class, search for the term `System.String` in the .NET Framework Class Library, as described in “To Learn More About the .NET Framework” on page 15-21.

The `string` function converts `String.String` arrays (`String.String[]`, `String.String[,]`, etcetera) to MATLAB string arrays with the same dimensions and sizes. Conversion of jagged arrays, for example `String.String[][]`, is not supported.

How MATLAB Handles System.__ComObject

The `System.__ComObject` type represents a Microsoft COM object. It is a non-visible, public class in the `mscorlib` assembly with no public methods. Under certain

circumstances, a .NET object returns an instance of `System.__ComObject`. MATLAB handles the `System.__ComObject` based on the return types defined in the metadata.

MATLAB Converts Object

If the return type of a method or property is strongly typed, and the result of the invocation is `System.__ComObject`, MATLAB automatically converts the returned object to the appropriate type.

For example, suppose that your assembly defines a type, `TestType`, and provides a method, `GetTestType`, with the following signature.

Return Type	Name	Arguments
<code>NetDocTest.TestType</code> <code>RetVal</code>	<code>GetTestType</code>	<code>(NetDocTest.MyClass this)</code>

The return type of `GetTestType` is strongly typed and the .NET Framework returns an object of type `System.__ComObject`. MATLAB automatically converts the object to the appropriate type, `NetDocTest.TestType`, shown in the following **pseudo-code**:

```
cls = NetDocTest.MyClass;
var = GetTestType(cls)

var =

    TestType handle with no properties.
```

Casting Object to Appropriate Type

If the return type of a method or property is `System.Object`, and the result of the invocation is `System.__ComObject`, MATLAB returns `System.__ComObject`. To use the returned object, cast it to a valid class or interface type. Use your product documentation to identify the valid types for this object.

To call a member of the new type, cast the object using the MATLAB conversion syntax:

```
objConverted = namespace.classname(obj)
```

where `obj` is a `System.__ComObject` type.

For example, an item in a Microsoft Excel sheet collection can be a chart or a worksheet. The following command converts the `System.__ComObject` variable `mySheet` to a `Chart` or a `Worksheet` object `newSheet`:

```
newSheet = Microsoft.Office.Interop.Excel.interfacename(mySheet);
```

where *interfacename* is `Chart` or `Worksheet`. For an example, see “Work with Microsoft Excel Spreadsheets Using .NET” on page 15-13.

Pass a COM Object Between Processes

If you pass a COM object to or from a function, lock the object so that MATLAB does not automatically release it when the object goes out of scope. To lock the object, call the `NET.disableAutoRelease` function. Then unlock the object, using the `NET.enableAutoRelease` function, after you are through using it.

How MATLAB Handles System.Nullable

If .NET returns a `System.Nullable` type, MATLAB returns the corresponding `System.Nullable` type.

A `System.Nullable` type lets you assign null values to types, such as numeric types, that do not support null value. To use a `System.Nullable` object in MATLAB, first decide how to handle null values.

- If you want to process null values differently from `<ValueType>` values, use the `HasValue` property.
- If you want every value to be of the underlying `<ValueType>`, use the `GetValueOrDefault` method. This method assigns a default value of type `<ValueType>` to null values.

Use a variable of the object's underlying type where appropriate in any MATLAB expression. For examples, see “Pass System.Nullable Arguments” on page 15-36.

How MATLAB Handles dynamic Type

MATLAB handles dynamic types as `System.Object`. For example, the following C# method `exampleMethod` has a dynamic input argument `d` and returns a dynamic output value:

```
public dynamic exampleMethod(dynamic d)
```

The following table shows the corresponding MATLAB function signature.

Return Type	Name	Arguments
System.Object RetVal	exampleMethod	(<i>namespace.classname</i> this, System.Object d)

How MATLAB Handles Jagged Arrays

You must convert a .NET jagged array before using it in a MATLAB command. To convert:

- If the shape of the array is rectangular, use the corresponding MATLAB numeric function.
- If the array is not rectangular, use the `cell` function.

If the jagged array is multidimensional, you must individually convert the arrays in each dimension.

Use Arrays with .NET Applications

In this section...

“Pass MATLAB Arrays to .NET” on page 15-63

“Convert Primitive .NET Arrays to MATLAB Arrays” on page 15-63

“Access .NET Array Elements in MATLAB” on page 15-64

“Convert .NET Jagged Arrays to MATLAB Arrays” on page 15-65

Pass MATLAB Arrays to .NET

MATLAB automatically converts arrays to .NET types, as described in the MATLAB Primitive Type Conversion Table. To pass an array of character arrays, create a cell array. For all other types, use the MATLAB NET.`createArray` function.

MATLAB creates a .NET array, copies the elements from the MATLAB array to the .NET array, and passes it to C#.

Convert Primitive .NET Arrays to MATLAB Arrays

To use a .NET array in MATLAB, call the appropriate MATLAB conversion function as shown in “Convert Arrays of Primitive .NET Type to MATLAB Type” on page 15-58. For example, suppose that a .NET method returns `netArr` of type `System.Int32[]`:

```
netArr =
  Int32[] with properties:
    Length: 5
    LongLength: 5
    Rank: 1
    SyncRoot: [1x1 System.Int32[]]
    IsReadOnly: 0
    IsFixedSize: 1
    IsSynchronized: 0
```

Convert the array to a MATLAB array of `int32`.

```
B = int32(netArr)
```

```
B = 1x5 int32 row vector
    1    2    3    4    5
```

Combine the elements in B with a MATLAB array.

```
A = int32([11 12 13 14 15]);  
A + B  
  
ans = 1x5 int32 row vector  
    12    14    16    18    20
```

Access .NET Array Elements in MATLAB

You access elements of a .NET array with subscripts, just like with MATLAB arrays.

You cannot refer to the elements of a multidimensional .NET array with a single subscript (linear indexing) like you can in MATLAB, as described in “Array Indexing”. You must specify the index for each dimension of the .NET array.

You can only use scalar indexing to access elements of a .NET array. The colon operator, described in “Creating, Concatenating, and Expanding Matrices”, is not supported.

Using the Get and Set Instance Functions

Alternatively, you can access elements of a .NET array using the `Set` and `Get` instance functions. When using `Set` or `Get` you must use C# array indexing, which is zero-based.

For example, create two `System.String` arrays, using the `Set` function and direct assignment:

```
d1 = NET.createArray('System.String',3);  
d1.Set(0, 'one');  
d1.Set(1, 'two');  
d1.Set(2, 'three');  
  
d2 = NET.createArray('System.String',3);  
d2(1) = 'one';  
d2(2) = 'two';  
d2(3) = 'zero';
```

To compare the values of the first elements in each array, type:

```
System.String.Compare(d1(1),d2.Get(0))
```

MATLAB displays `0`, meaning the strings are equal.

Convert .NET Jagged Arrays to MATLAB Arrays

You must convert a .NET jagged array before using it in a MATLAB command.

- If the shape of the array is rectangular, use the corresponding MATLAB numeric function.
- If the array is not rectangular, use the `cell` function.

If the jagged array is multidimensional, you must individually convert the arrays in each dimension.

See Also

`NET.createArray` | `cell`

More About

- “Convert .NET Arrays to Cell Arrays” on page 15-66
- “Limitations to Support of .NET Arrays” on page 15-69

Convert .NET Arrays to Cell Arrays

In this section...

“Convert Nested System.Object Arrays” on page 15-66

“cell Function Syntax for System.Object[,] Arrays” on page 15-67

To convert .NET `System.String` and `System.Object` arrays to MATLAB cell arrays, use the `cell` function. Elements of the cell array are of the MATLAB type closest to the .NET type. For more information, see “.NET Type to MATLAB Type Mapping” on page 15-57.

For example, use the .NET Framework `System.IO.Directory` class to create a cell array of folder names in your `c:\` folder.

```
myList = cell(System.IO.Directory.GetDirectories('c:\'));
```

Convert Nested System.Object Arrays

The conversion is not recursive for a `System.Object` array contained within a `System.Object` array. You must use the `cell` function to convert each `System.Object` array.

For an example, build the `NetDocCell` assembly using the directions in “Build a .NET Application for MATLAB Examples” on page 15-24. The source code is here.

C# NetDocCell Source File

```
using System;
/*
 * C# Assembly used in MATLAB .NET documentation.
 * Method getNewData is used to demonstrate
 * how MATLAB handles a System.Object
 * that includes another System.Object.
 */
namespace NetDocCell
{
    public class MyGraph
    {
        public Object[] getNewData()
        /*
         * Create a System.Object array to use in MATLAB examples.
         */
    }
}
```

```

    * Returns containerArr System.Object array containing:
    * fLabel System.String object
    * plotData System.Object array containing:
    *   xLabel System.String object
    *   doubleArr System.Double array
    */
    {
        String fLabel = "Figure Showing New Graph Data";
        Double[] doubleArr = {
            18, 32, 3.133, 44, -9.9, -13, 33.03 };
        String xLabel = "X-Axis Label";
        Object[] plotData = { xLabel, doubleArr };
        Object[] containerArr = { fLabel, plotData };
        return containerArr;
    }
}

```

Load the assembly and create a cell array, `mlData`.

```

dllPath = fullfile('c:', 'work', 'NetDocCell.dll');
NET.addAssembly(dllPath);
obj = NetDocCell.MyGraph;
mlData = cell(obj.getNewData)

```

The cell array contains elements of type

```

mlData =
    [1x1 System.String]    [1x1 System.Object[]]

```

To access the contents of the `System.Object` array, create another cell array `mlPlotData`.

```

mlPlotData = cell(mlData{2})

```

This cell array contains elements of type

```

mlPlotData =
    [1x1 System.String]    [1x1 System.Double[]]

```

cell Function Syntax for System.Object[,] Arrays

Use this `cell` function syntax to convert `System.DateTime` and `System.String` data contained in a `System.Object[,]` array to cell arrays of MATLAB data,

```
A = cell(obj, 'ConvertTypes', type)
```

where `obj` is a `.NET System.Object[,]` array, and `type` is one of the following:

- `{'System.DateTime'}` — Convert `System.DateTime` elements to MATLAB `datetime` elements.
- `{'System.String'}` — Convert `System.String` elements to MATLAB character arrays.
- `{'all'}` — Convert all supported .NET types to equivalent MATLAB types.

`A` is a cell array, that is the same size as the `obj` array.

See Also

Related Examples

- “Pass Cell Arrays of .NET Data” on page 15-40
- “Read Cell Arrays of Excel Spreadsheet Data” on page 15-4

Limitations to Support of .NET Arrays

MATLAB does not support:

- Arrays which specify a lower bound
- Concatenating .NET objects into an array
- The `end` function as the last index in a .NET array
- Array indices of complex values
- Autoconversion of `char` or `cell` arrays to jagged array arguments.
- Autoconversion of MATLAB arrays to multidimensional jagged array arguments.

Set Static .NET Properties

In this section...
“Set System.Environment.CurrentDirectory Static Property” on page 15-70
“Do Not Use ClassName.PropertyName Syntax for Static Properties” on page 15-71

Set System.Environment.CurrentDirectory Static Property

This example shows how to set a static property using the `NET.setStaticProperty` function.

The `CurrentDirectory` property in the `System.Environment` class is a static, read/write property. The following code creates a folder `temp` in the current folder and changes the `CurrentDirectory` property to the new folder.

Set your current folder.

```
cd('C:\Work')
```

Set the `CurrentDirectory` property.

```
saveDir = System.Environment.CurrentDirectory;  
newDir = [char(saveDir) '\temp'];  
mkdir(newDir)  
NET.setStaticProperty('System.Environment.CurrentDirectory',newDir)  
System.Environment.CurrentDirectory
```

```
ans =
```

```
C:\Work\temp
```

Restore the original `CurrentDirectory` value.

```
NET.setStaticProperty('System.Environment.CurrentDirectory',saveDir)
```

Do Not Use ClassName.PropertyName Syntax for Static Properties

This example shows how to mistakenly create a `struct` array instead of setting a class property.

If you use the `ClassName.PropertyName` syntax to set a static property, MATLAB creates a `struct` array.

The following code creates a structure named `System`:

```
saveDir = System.Environment.CurrentDirectory;  
newDir = [char(saveDir) '\\temp'];  
System.Environment.CurrentDirectory = newDir;  
whos
```

Name	Size	Bytes	Class
System	1x1	376	struct
newDir	1x12	24	char
saveDir	1x1	112	System.String

Try to use a member of the `System` namespace.

```
oldDate = System.DateTime(1992,3,1);
```

Reference to non-existent field 'DateTime'.

To restore your environment, type:

```
clear System  
NET.setStaticProperty('System.Environment.CurrentDirectory',saveDir)
```

Using .NET Properties

In this section...

“How MATLAB Represents .NET Properties” on page 15-72

“How MATLAB Maps C# Property and Field Access Modifiers” on page 15-72

How MATLAB Represents .NET Properties

To view property names, use the `properties` function.

To get and set the value of a class property, use the MATLAB dot notation:

```
x = ClassName.PropertyName;  
ClassName.PropertyName = y;
```

The following example gets the value of a property (the current day of the month):

```
dtnow = System.DateTime.Now;  
d = dtnow.Day;
```

The following example sets the value of a property (the Volume for a `SpeechSynthesizer` object):

```
NET.addAssembly('System.Speech');  
ss = System.Speech.Synthesis.SpeechSynthesizer;  
ss.Volume = 50;  
Speak(ss, 'You can use .NET Libraries in MATLAB')
```

To set a static property, call the `NET.setStaticProperty` function. For an example, see “Set Static .NET Properties” on page 15-70.

MATLAB represents public .NET fields as properties.

MATLAB represents .NET properties that take an argument as methods. For more information, see “Call .NET Properties That Take an Argument” on page 15-90.

How MATLAB Maps C# Property and Field Access Modifiers

MATLAB maps C# keywords to MATLAB property attributes, as shown in the following table.

C# Property Keyword	MATLAB Attribute
public, static	Access = public
protected, private, internal	Not visible to MATLAB
get, set	Access = public
Get	GetAccess = public, SetAccess = private
Set	SetAccess = public, GetAccess = private

MATLAB maps C# keywords to MATLAB field attributes, as shown in the following table.

C# Field Keyword	MATLAB Mapping
public	Supported
protected, private, internal, protected internal	Not visible to MATLAB

For more information about MATLAB properties, see “Property Attributes”.

MATLAB Does Not Display Protected Properties

The `System.Windows.Media.ContainerVisual` class, available in .NET Framework Version 3.0 and above, has several protected properties. MATLAB only displays public properties and fields. Type:

```
NET.addAssembly('PresentationCore');  
properties('System.Windows.Media.ContainerVisual')
```

Display Public Properties

Properties for class `System.Windows.Media.ContainerVisual`:

```
Children  
Parent  
Clip  
Opacity  
OpacityMask  
CacheMode  
BitmapEffect  
BitmapEffectInput  
Effect  
XSnappingGuidelines  
YSnappingGuidelines  
ContentBounds  
Transform  
Offset  
DescendantBounds  
DependencyObjectType  
IsSealed  
Dispatcher
```

To see how MATLAB handles property and field C# keywords, see “How MATLAB Maps C# Property and Field Access Modifiers” on page 15-72.

Work with .NET Methods Having Multiple Signatures

To create the `NetSample` assembly, see “Build a .NET Application for MATLAB Examples” on page 15-24.

The `SampleMethodSignature` class defines the three constructors shown in the following table.

Return Type	Name	Arguments
<code>netdoc.SampleMethodSignature obj</code>	<code>SampleMethodSignature</code>	
<code>netdoc.SampleMethodSignature obj</code>	<code>SampleMethodSignature</code>	<code>(double scalar d)</code>
<code>netdoc.SampleMethodSignature obj</code>	<code>SampleMethodSignature</code>	<code>(System.String s)</code>

SampleMethodSignature Class

```
using System;
namespace netdoc
{
    public class SampleMethodSignature
    {
        public SampleMethodSignature ()
        {}

        public SampleMethodSignature (double d)
        { myDoubleField = d; }

        public SampleMethodSignature (string s)
        { myStringField = s; }

        public int myMethod(string strIn, ref double dbRef,
                            out double dbOut)
        {
            dbRef += dbRef;
            dbOut = 65;
            return 42;
        }

        private Double myDoubleField = 5.5;
    }
}
```

```
        private String myStringField = "hello";  
    }  
}
```

Display Function Signature Example

If you have not already loaded the `NetSample` assembly, type:

```
NET.addAssembly('c:\work\NetSample.dll')
```

Create a `SampleMethodSignature` object `obj`:

```
obj = netdoc.SampleMethodSignature;
```

To see the method signatures, type:

```
methods(obj, '-full')
```

Look for the following signatures in the MATLAB output:

```
netdoc.SampleMethodSignature obj SampleMethodSignature  
netdoc.SampleMethodSignature obj SampleMethodSignature(double scalar d)  
netdoc.SampleMethodSignature obj SampleMethodSignature(System.String s)
```

For more information about argument types, see “Handle Data Returned from .NET Objects” on page 15-57.

Call .NET Methods With out Keyword

This example shows how to call methods that use an out keyword in the argument list.

The output argument db2 in the following outTest method is modified by the out keyword.

```
using System;
namespace netdoc
{
    public class SampleOutTest
    {
        //test out keyword
        public void outTest(double db1, out double db2)
        {
            db1 = db1 * 2.35;
            db2 = db1;
        }
    }
}
```

The function signature in MATLAB is:

Return Type	Name	Arguments
double scalar db2	outTest	(netdoc.SampleOutTest this, double scalar db1)

Create an assembly from the SampleOutTest code, using instructions in Build a .NET Application for MATLAB Examples.

Create an asmpath variable set to the full path to the DLL file, SampleOutTest.dll, created by your development tool. For example:

```
asmpath = 'c:\work\Visual Studio 2012\Projects\SampleOutTest\SampleOutTest\bin\Debug\'
asmname = 'SampleOutTest.dll';
```

Load the assembly.

```
asm = NET.addAssembly(fullfile(asmpath,asmname));
```

Call the method.

```
cls = netdoc.SampleOutTest;  
db3 = outTest(cls,6)  
  
db3 =  
    14.1000
```

See Also

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24

More About

- “C# Method Access Modifiers” on page 15-86

Call .NET Methods With ref Keyword

This example shows how to call methods that use a `ref` keyword in the argument list.

The input argument `dbl` in the following `refTest` method is modified by the `ref` keyword.

```
using System;
namespace netdoc
{
    public class SampleRefTest
    {
        //test ref keyword
        public void refTest(ref double dbl)
        {
            dbl = dbl * 2;
        }
    }
}
```

The function signature in MATLAB is:

Return Type	Name	Arguments
double scalar dbl	refTest	(netdoc.SampleRefTest this, double scalar dbl)

Create an assembly from the `SampleRefTest` code, using instructions in Build a .NET Application for MATLAB Examples.

Create an `asmpath` variable set to the full path to the DLL file, `SampleRefTest.dll`, created by your development tool. For example:

```
asmpath = 'c:\work\Visual Studio 2012\Projects\SampleRefTest\SampleRefTest\bin\Debug\';
asmname = 'SampleRefTest.dll';
```

Load the assembly.

```
asm = NET.addAssembly(fullfile(asmpath,asmname));
```

Call the method.

```
cls = netdoc.SampleRefTest;
dbl = refTest(cls,6)
```

```
db4 =  
    12
```

See Also

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24

More About

- “C# Method Access Modifiers” on page 15-86

Call .NET Methods With params Keyword

This example shows how to call methods that use a params keyword in the argument list.

The input argument num in the following paramsTest method is modified by the params keyword.

```
using System;
namespace netdoc
{
    public class SampleParamsTest
    {
        //test params keyword
        public int paramsTest(params int[] num)
        {
            int total = 0;
            foreach (int i in num)
            {
                total = total + i;
            }
            return total;
        }
    }
}
```

The function signature in MATLAB is:

Return Type	Name	Arguments
int32 scalar RetVal	paramsTest	(netdoc.SampleParamsTest this, System.Int32[] num)

Create an assembly from the SampleParamsTest code, using instructions in Build a .NET Application for MATLAB Examples.

Create an asmpath variable set to the full path to the DLL file, SampleParamsTest.dll, created by your development tool. For example:

```
asmpath = 'c:\work\Visual Studio 2012\Projects\SampleParamsTest\SampleParamsTest\bin\Debug\';
asmname = 'SampleParamsTest.dll';
```

Load the assembly.

```
asm = NET.addAssembly(fullfile(asmpath,asmname));
```

Call the method.

```
cls = netdoc.SampleParamsTest;  
mat = [1, 2, 3, 4, 5, 6];  
db5 = paramsTest(cls,mat)
```

```
db5 =  
      21
```

See Also

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24

More About

- “C# Method Access Modifiers” on page 15-86

Call .NET Methods with Optional Arguments

In this section...

“Skip Optional Arguments” on page 15-83

“Call Overloaded Methods” on page 15-84

Skip Optional Arguments

This example shows how to use default values in optional arguments using the `Greeting` method.

Greeting Function Signature

Arguments `str1` and `str2` are optional.

Return Type	Name	Arguments
System.StringRetVal	Greeting	(NetDocOptional.MyClass this, int32 scalar x, optional<System.String> str1, optional<System.String> str2)

Build the C# example, `NetDocOptional.cs` in the `matlabroot/extern/examples/NET/NetSample` folder. For information about building the assembly, see “Build a .NET Application for MATLAB Examples” on page 15-24.

Load the `NetDocOptional` assembly, if it is not already loaded.

```
dllPath = fullfile('c:', 'work', 'NetDocOptional.dll');
asm = NET.addAssembly(dllPath);
cls = NetDocOptional.MyClass;
```

The example assumes that you put the assembly in your `c:\work` folder. You can modify the `fullfile` function to change the path to the assembly.

Display the default values.

```
Greeting(cls,0)
```

```
ans =
hello world
```

Use the default value for `str1`.

```
def = System.Reflection.Missing.Value;
Greeting(cls,0,def,'Mr. Jones')
```

```
ans =
hello Mr. Jones
```

Use the default value for `str2`. You can omit the argument at the end of a parameter list.

```
Greeting(cls,0,'My')
```

```
ans =
My world
```

Call Overloaded Methods

This example shows how to use optional arguments with an overloaded method, `calc`. To run the example, you must create and build your own assembly `Doc` defining the `calc` method in `Class` with the following function signatures.

calc Function Signatures

The following table shows the signatures for `calc`, which adds the input arguments. The difference is the type of optional argument, `y`.

Return Type	Name	Arguments
single scalar RetVal	calc	(Doc.Class this, optional<int32 scalar> x, optional<single scalar> y)
double scalar RetVal	calc	(Doc.Class this, optional<int32 scalar> x, optional<double scalar> y)

Load your assembly and create `cls`.

```
cls = Doc.Class;
```

Call `calc` with explicit arguments.

```
calc(cls,3,4)
```

```
ans =  
    7
```

If you try to use the default values by omitting the parameters, MATLAB cannot determine which signature to use.

```
calc(cls)
```

Cannot choose between the following .NET method signatures due to unspecified optional arguments in the call to 'calc':

```
'Doc.Class.calc(Doc.Class this,  
optional<int32 scalar> x, optional<single scalar> y)' and  
'Doc.Class.calc(Doc.Class this,  
optional<int32 scalar> x, optional<double scalar> y)'
```

You can resolve this ambiguity by specifying enough additional optional arguments so that there is only one possible matching .NET method.

To use the default values, you must provide both arguments.

```
def = System.Reflection.Missing.Value;  
calc(cls,def,def)  
calc(cls,3,def)  
calc(cls,def,4)
```

```
ans =  
    44  
ans =  
    14  
ans =  
    37
```

Calling .NET Methods

In this section...

“Getting Method Information” on page 15-86

“C# Method Access Modifiers” on page 15-86

“VB.NET Method Access Modifiers” on page 15-87

“Reading Method Signatures” on page 15-87

Getting Method Information

Use the following MATLAB functions to view the methods of a class. You can use these functions without creating an instance of the class. These functions do not list generic methods; use your product documentation to get information on generic methods.

- `methods` — View method names
- `methods` with ' - full ' option — View method names with argument list
- `methodsview` — Graphical representation of method list

You might find the `methodsview` window easier to use as a reference guide because you do not need to scroll through the Command Window to find information. For example, open a `methodsview` window for the `System.String` class:

```
methodsview('System.String')
```

C# Method Access Modifiers

MATLAB maps C# keywords to MATLAB method access attributes, as shown in the following table.

C# Method Keyword	MATLAB Attribute	Example
ref	RHS, LHS	“Call .NET Methods With ref Keyword” on page 15-79
out	LHS	“Call .NET Methods With out Keyword” on page 15-77

C# Method Keyword	MATLAB Attribute	Example
params	Array of particular type	"Call .NET Methods With params Keyword" on page 15-81
protected, private, internal, protected internal	Not visible to MATLAB	

VB.NET Method Access Modifiers

MATLAB maps VB.NET keywords to MATLAB method access attributes, as shown in the following table.

VB.NET Method Keyword	MATLAB Attribute
ByRef	LHS, RHS
ByVal	RHS
Optional	Mandatory

Reading Method Signatures

MATLAB uses the following rules to populate method signatures.

- `obj` is the output from the constructor.
- `this` is the object argument.
- `RetVal` is the return type of a method.
- All other arguments use the .NET metadata.

MATLAB uses the following rules to select a method signature.

- Number of inputs
- Input type
- Number of outputs

Calling .NET Methods with Optional Arguments

MATLAB displays optional arguments in a method signature using the `optional<T>` syntax, where T is the specific type. This feature is available in .NET Framework Version 4.0 and above.

To use a default method argument, pass an instance of `System.Reflection.Missing.Value`.

Skipping Optional Arguments

If the method is not overloaded, you are not required to fill in all optional values at the end of a parameter list. For examples, see “Skip Optional Arguments” on page 15-83.

Determining Which Overloaded Method Is Invoked

If a .NET class has overloaded methods with optional arguments, MATLAB picks the method matching the exact number of input arguments.

If the optional arguments of the methods are different by type, number, or dimension, MATLAB first compares the types of the mandatory arguments. If the types of the mandatory arguments are different, MATLAB chooses the first overloaded method defined in the class. If the types of the mandatory arguments are the same, specify enough optional arguments so that there is only one possible matching .NET method. Otherwise, MATLAB throws an error. For examples, see “Call Overloaded Methods” on page 15-84.

Support for ByRef Attribute in VB.NET

The rules for optional ByRef arguments are the same as for other method arguments, as described in “VB.NET Method Access Modifiers” on page 15-87. ByRef arguments on the RHS appear as optional and behave like any other optional argument.

Calling .NET Extension Methods

Unlike C# applications, MATLAB handles an extension method as a static method of the class that defines the method. Refer to your product documentation for the namespace and class name you need to call such methods.

For information about extension methods, see the MSDN article at [https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/bb383977\(v=vs.90\)](https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/bb383977(v=vs.90)).

Call .NET Properties That Take an Argument

MATLAB represents a property that takes an argument as a method. For example, the `System.String` class has two properties, `Chars` and `Length`. The `Chars` property gets the character at a specified character position in the `System.String` object. For example:

```
str = System.String('my new string');  
methods(str)
```

Display of System.String Methods

Methods for class `System.String`:

<code>Chars</code>	<code>Normalize</code>	<code>TrimStart</code>
<code>Clone</code>	<code>PadLeft</code>	<code>addListener</code>
<code>CompareTo</code>	<code>PadRight</code>	<code>char</code>
<code>Contains</code>	<code>Remove</code>	<code>delete</code>
<code>CopyTo</code>	<code>Replace</code>	<code>eq</code>
<code>EndsWith</code>	<code>Split</code>	<code>findobj</code>
<code>Equals</code>	<code>StartsWith</code>	<code>findprop</code>
<code>GetEnumerator</code>	<code>String</code>	<code>ge</code>
<code>GetHashCode</code>	<code>Substring</code>	<code>gt</code>
<code>GetType</code>	<code>ToCharArray</code>	<code>invalid</code>
<code>GetTypeCode</code>	<code>ToLower</code>	<code>le</code>
<code>IndexOf</code>	<code>ToLowerInvariant</code>	<code>lt</code>
<code>IndexOfAny</code>	<code>ToString</code>	<code>ne</code>
<code>Insert</code>	<code>ToUpper</code>	<code>notify</code>
<code>IsNormalized</code>	<code>ToUpperInvariant</code>	
<code>LastIndexOf</code>	<code>Trim</code>	
<code>LastIndexOfAny</code>	<code>TrimEnd</code>	

Static methods:

<code>Compare</code>	<code>Intern</code>	<code>op_Equality</code>
<code>CompareOrdinal</code>	<code>IsInterned</code>	<code>op_Inequality</code>
<code>Concat</code>	<code>IsNullOrEmpty</code>	
<code>Copy</code>	<code>IsNullOrWhiteSpace</code>	
<code>Format</code>	<code>Join</code>	

Notice that MATLAB displays the `Chars` property as a method.

The `Chars` method has the following signature.

Return Type	Name	Arguments
char scalar RetVal	Chars	(System.String this, int32 scalar index)

To see the first character, type:

```
Chars(str,0)
```

```
ans =  
m
```

How MATLAB Represents .NET Operators

MATLAB supports overloaded operators, such as the C# operator symbols + and *, as shown in the following table. MATLAB implements all other overloaded operators, such as % and +=, by their static method names, `op_Modulus` and `op_AdditionAssignment`. For a complete list of operator symbols and the corresponding operator names, see [https://msdn.microsoft.com/en-us/library/2sk3x8a7\(VS.71\).aspx](https://msdn.microsoft.com/en-us/library/2sk3x8a7(VS.71).aspx) on the Microsoft Developer Network website.

C++ Operator Symbol	.NET Operator	MATLAB Method
+ (binary)	<code>op_Addition</code>	plus, +
- (binary)	<code>op_Subtraction</code>	minus, -
* (binary)	<code>op_Multiply</code>	mtimes, *
/	<code>op_Division</code>	mrdivide, /
&&	<code>op_LogicalAnd</code>	and, &
	<code>op_LogicalOr</code>	or,
==	<code>op_Equality</code>	eq, ==
>	<code>op_GreaterThan</code>	gt, >
<	<code>op_LessThan</code>	lt, <
!=	<code>op_Inequality</code>	ne, ~=
>=	<code>op_GreaterThanOrEqual</code>	ge, >=
<=	<code>op_LessThanOrEqual</code>	le, <=
- (unary)	<code>op_UnaryNegation</code>	uminus, -a
+ (unary)	<code>op_UnaryPlus</code>	uplus, +a

Limitations to Support of .NET Methods

Displaying Generic Methods

The `methods` and `methodsview` functions do not list generic methods.

Overloading MATLAB Functions

If your application implements a method with the same name as a MATLAB function, the method must have the same signature as the MATLAB function. Otherwise, MATLAB throws an error. For information about how MATLAB handles overloaded functions, see the following topics:

- “Overload Functions in Class Definitions”
- “Methods That Modify Default Behavior”

Calling Overloaded Static Methods

If an instance method has the same name as a static method, then MATLAB throws an error when you call the instance method.

Use .NET Events in MATLAB

These examples use the `addListener` function to handle .NET events with MATLAB callbacks.

Monitor Changes to .TXT File

This example uses the `System.IO.FileSystemWatcher` class in the `System` assembly to monitor changes to a .TXT file in the `C:\work\temp` folder. Create the following event handler, `eventhandlerChanged.m`:

```
function eventhandlerChanged(source, arg)
disp('TXT file changed')
end
```

Create a `FileSystemWatcher` object `fileObj` and watch the `Changed` event for files with a `.txt` extension in the folder `C:\work\temp`.

```
file = System.IO.FileSystemWatcher('c:\work\temp');
file.Filter = '*.txt';
file.EnableRaisingEvents = true;
addListener(file, 'Changed', @eventhandlerChanged);
```

If you modify and save a `.txt` file in the `C:\work\temp` folder, MATLAB displays:

```
TXT file changed
```

The `FileSystemWatcher` documentation says that a simple file operation can raise multiple events.

To turn off the event handler, type:

```
file.EnableRaisingEvents = false;
```

Monitor Changes to Windows Form ComboBox

This example shows how to listen for changes to values in a `ComboBox` on a Windows Form. This example uses the `SelectedValueChanged` event defined by the `System.Windows.Forms.ComboBox` class.

To create this example, you must build a Windows Forms Application using a supported version of Microsoft Visual Studio.

- Search the Microsoft MSDN website for information about Windows Forms Applications.
- For an up-to-date list of supported compilers, see Supported and Compatible Compilers.

Create a 64-bit Windows Forms Application, `myForm`, in your `C:\work` folder. Add a `ComboBox` control to `Form1`, and then add one or more items to `ComboBox1`. Build the application.

To add a listener to the form property, create the following MATLAB class, `EnterComboData`, which uses the `attachListener` method.

```
classdef EnterComboData < handle
properties
    form
end
methods
    function x = EnterComboData
        NET.addAssembly('C:\work\myForm\myForm\bin\x64\Debug\myForm.exe');
        x.form = myForm.Form1;
        Show(x.form)
        Activate(x.form)
    end
    function r = attachListener(x)
        % create listener
        r = addlistener(
            x.form.Controls.Item(0),
            'SelectedValueChanged',
            @x.anyChange);
    end
    function anyChange(~,~,~)
        % listener action if comboBox changes
        disp('Field updated')
    end
end
end
```

To execute the following MATLAB commands, you must create and load the application named `myForm.exe`. To create a form and call its `attachListener` method, use the `EnterComboData` class.

```
form = EnterComboData;
form.attachListener;
```

To trigger an event, select an item from the drop-down menu on the ComboBox. MATLAB displays:

```
Field updated
```

See Also

`addlistener`

More About

- “Limitations to Support of .NET Events” on page 15-108

Call .NET Delegates in MATLAB

This example shows you how to use a delegate in MATLAB. It creates a delegate using a MATLAB function (`char`).

Declare a Delegate in a C# Assembly

The C# example `NetDocDelegate.cs`, in the `matlabroot/extern/examples/NET/NetSample` folder, defines delegates used in the following examples. To see the code, open the file in MATLAB Editor. To run the examples, build the `NetDocDelegate` assembly as described in “Build a .NET Application for MATLAB Examples” on page 15-24.

Load the Assembly Containing the Delegate into MATLAB

If the `NetDocDelegate` assembly is in your `c:\work` folder, load the file with the command:

```
dllPath = fullfile('c:', 'work', 'NetDocDelegate.dll');  
NET.addAssembly(dllPath);
```

Select a MATLAB Function

The `delInteger` delegate encapsulates any method that takes an integer input and returns a string. The MATLAB `char` function, which converts a nonnegative integer into a character array, has a signature that matches the `delInteger` delegate. For example, the following command displays the `!` character:

```
char(33)
```

Create an Instance of the Delegate in MATLAB

To create an instance of the `delInteger` delegate, pass the function handle of the `char` function:

```
myFunction = NetDocDelegate.delInteger(@char);
```

Invoke the Delegate Instance in MATLAB

Use `myFunction` the same as you would `char`. For example, the following command displays the `!` character:

```
myFunction(33)
```

See Also

`char`

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24
- “Create Delegates from .NET Object Methods” on page 15-99

More About

- “Limitations to Support of .NET Delegates” on page 15-109
- “.NET Delegates” on page 15-106

Create Delegates from .NET Object Methods

The following C# class defines the methods `AddEggs` and `AddFlour`, which have signatures matching the `delInteger` delegate:

C# Recipe Source File

```
using System;
namespace Recipe
{
    public class MyClass
    {
        public string AddEggs(double n)
        {
            return "Add " + n + " eggs";
        }

        public string AddFlour(double n)
        {
            return "Add " + n + " cups flour";
        }
    }
}
```

Build the `Recipe` assembly, and then load it and create a delegate `myFunc` using `AddEggs` as the callback:

```
NET.addAssembly(dllPath);
NET.addAssembly('c:\work\Recipe.dll');
myRec = Recipe.MyClass;
myFunc = NetDocDelegate.delInteger(@myRec.AddEggs);
myFunc(2)

ans =
Add 2 eggs
```

See Also

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24

- “Call .NET Delegates in MATLAB” on page 15-97

More About

- “Limitations to Support of .NET Delegates” on page 15-109

Create Delegate Instances Bound to .NET Methods

For a C# delegate defined as:

```
namespace MyNamespace
{
    public delegate void MyDelegate();
}
```

MATLAB creates the following constructor signature.

Return Type	Name	Arguments
MyNamespace.MyDelegate obj	MyDelegate	(target, string methodName)

The argument `target` is one of the following:

- An instance of the invocation target object when binding to the instance method
- A string with fully qualified .NET class name when binding to a static method

`methodName` is a string specifying the callback method name.

Example — Create a Delegate Instance Associated with a .NET Object Instance Method

For the following C# delegate and class definition:

```
namespace MyNamespace
{
    public delegate void MyDelegate();

    public class MyClass
    {
        public void MyMethod(){}
    }
}
```

To instantiate the delegate in MATLAB, type:

```
target = MyNamespace.MyClass();
delegate = MyNamespace.MyDelegate(target, 'MyMethod');
```

Example — Create a Delegate Instance Associated with a Static .NET Method

For the following C# delegate and class definition:

```
namespace MyNamespace
{
    public delegate void MyDelegate();

    public class MyClass
    {
        public static void MyStaticMethod(){}
    }
}
```

To instantiate the delegate in MATLAB, type:

```
delegate = MyNamespace.MyDelegate('MyNamespace.MyClass', 'MyStaticMethod');
```

See Also

Related Examples

- “Build a .NET Application for MATLAB Examples” on page 15-24

.NET Delegates With out and ref Type Arguments

The MATLAB rules for mapping out and ref types for delegates are the same as for methods.

For example, the following C# statement declares a delegate with a ref argument:

```
public delegate void delref(ref Double refArg);
```

The signature for an equivalent MATLAB delegate function maps refArg as both RHS and LHS arguments:

```
function refArg = myFunc(refArg)
```

The following C# statement declares a delegate with an out argument:

```
public delegate void delout(  
    Single argIn,  
    out Single argOut);
```

The signature for an equivalent MATLAB delegate function maps argOut as an LHS argument:

```
function argOut = myFunc(argIn)
```

See Also

More About

- “C# Method Access Modifiers” on page 15-86

Combine and Remove .NET Delegates

MATLAB provides the instance method `Combine`, that lets you combine a series of delegates into a single delegate. The `Remove` and `RemoveAll` methods delete individual delegates. For more information, refer to the .NET Framework Class Library, as described in “To Learn More About the .NET Framework” on page 15-21.

For example, create the following MATLAB functions to use with the `NetDocDelegate.delInteger` delegate:

```
function out = action1(n)
out = 'Add flour';
disp(out)
end

function out = action2(n)
out = 'Add eggs';
disp(out)
end
```

Create delegates `step1` and `step2`:

```
step1 = NetDocDelegate.delInteger(@action1);
step2 = NetDocDelegate.delInteger(@action2);
```

To combine into a new delegate, `mixItems`, type:

```
mixItems = step1.Combine(step2);
```

Or, type:

```
mixItems = step1.Combine(@action2);
```

Invoke `mixItems`:

```
result = mixItems(1);
```

In this case, the function `action2` follows `action1`:

```
Add flour
Add eggs
```

The value of `result` is the output from the final delegate (`step2`).


```
result =  
Add eggs
```

You also can use the `System.Delegate` class static methods, `Combine`, `Remove`, and `RemoveAll`.

To remove a `step1` from `mixItems`, type:

```
step3 = mixItems.Remove(step1);
```

See Also

[Combine](#) | [Remove](#) | [RemoveAll](#)

Related Examples

- [“Call .NET Delegates in MATLAB”](#) on page 15-97

.NET Delegates

In the .NET Framework, a delegate is a type that defines a method signature. It lets you pass a function as a parameter. The use of delegates enables .NET applications to make calls into MATLAB callback functions or class instance methods. For the rules MATLAB uses to define the signature of a callback function or class method, see “Reading Method Signatures” on page 15-87. For a complete description of delegates and when to use them, consult an outside resource, such as the Microsoft Developer Network.

There are three steps to using delegates:

- Declaration — Your .NET application contains the declaration. You cannot declare a delegate in the MATLAB language.
- Instantiation — In MATLAB, create an instance of the delegate and associate it with a specific MATLAB function or .NET object method.
- Invocation — Call the function with specified input and output arguments. Use the delegate name in place of the function name.

See Also

Related Examples

- “Call .NET Delegates in MATLAB” on page 15-97

Calling .NET Methods Asynchronously

How MATLAB Handles Asynchronous Method Calls in .NET

It is possible to call a synchronous method asynchronously in MATLAB. With some modifications, you can use the Microsoft `BeginInvoke` and `EndInvoke` methods. For more information, refer to the MSDN article [Calling Synchronous Methods Asynchronously](#).

You can use delegates to call a synchronous method asynchronously by using the `BeginInvoke` and `EndInvoke` methods. If the thread that initiates the asynchronous call does not need to be the thread that processes the results, you can execute a callback method when the call completes.

Note MATLAB is a single-threaded application. Therefore, handling asynchronous calls in the MATLAB environment might result in deadlocks.

Using `EndInvoke` With `out` and `ref` Type Arguments

The MATLAB delegate signature for `EndInvoke` follows special mapping rules if your delegate has `out` or `ref` type arguments. For information about the mapping, see “.NET Delegates With `out` and `ref` Type Arguments” on page 15-103. For examples, see the `EndInvoke` reference page.

Using Polling to Detect When Asynchronous Call Finishes

For MATLAB to process the event that executes the delegate callback on the main thread, call the MATLAB `pause` (or a similar) function.

See Also

[BeginInvoke](#) | [EndInvoke](#)

External Websites

- [MSDN article “Calling Synchronous Methods Asynchronously”](#)

Limitations to Support of .NET Events

MATLAB Support of Standard Signature of an Event Handler Delegate

An event handler in C# is a delegate with the following signature:

```
public delegate void MyEventHandler(object sender, MyEventArgs e)
```

The argument `sender` specifies the object that fired the event. The argument `e` holds data that can be used in the event handler. The class `MyEventArgs` is derived from the .NET Framework class `EventArgs`. MATLAB only handles events with this standard signature.

Limitations to Support of .NET Delegates

MATLAB does not support associating a delegate instance with a generic .NET method.

When calling a method asynchronously, be aware that:

- MATLAB is a single-threaded application. Therefore, handling asynchronous calls in the MATLAB environment might result in deadlocks.
- For the technique described in the MSDN topic [Blocking Application Execution Using an AsyncWaitHandle](#), MATLAB does not support the use of the `WaitOne()` method overload with no arguments.
- You cannot call `EndInvoke` to wait for the asynchronous call to complete.

Use Bit Flags with .NET Enumerations

In this section...

“How MATLAB Supports Bitwise Operations on System.Enum” on page 15-110

“Creating .NET Enumeration Bit Flags” on page 15-110

“Removing a Flag from a Variable” on page 15-111

“Replacing a Flag in a Variable” on page 15-112

“Testing for Membership” on page 15-112

How MATLAB Supports Bitwise Operations on System.Enum

Many .NET languages support bitwise operations on enumerations defined with the `System.Flags` attribute. The MATLAB language does not have equivalent operations, and, therefore, provides instance methods for performing bitwise operations on an enumeration object. The bitwise methods are `bitand`, `bitnot`, `bitor`, and `bitxor`.

An enumeration can define a bit flag. A bit flag lets you create instances of an enumeration to store combinations of values defined by the members. For example, files and folders have attributes, such as `Archive`, `Hidden`, and `ReadOnly`. For a given file, perform an operation based on one or more of these attributes. With bitwise operators, you can create and test for combinations.

To use bitwise operators, the enumeration must have:

- The `Flags` attribute. In Framework Version 4, these enumerations also have the `HasFlag` method.
- Values that correspond to powers of 2.

Creating .NET Enumeration Bit Flags

Use the MATLAB example, `NetDocEnum.MyDays` enumeration, in the following examples.

Suppose that you have the following scheduled activities:

- Monday — Department meeting at 10:00
- Wednesday and Friday — Team meeting at 2:00

- Thursday — Volley ball night

You can combine members of the `MyDays` enumeration to create MATLAB variables using the `bitor` method, which joins two members. For example, to create a variable `teamMtgs` of team meeting days, type:

```
teamMtgs = bitor(...  
    NetDocEnum.MyDays.Friday,...  
    NetDocEnum.MyDays.Wednesday);
```

Create a variable `allMtgs` of all days with meetings:

```
allMtgs = bitor(teamMtgs,...  
    NetDocEnum.MyDays.Monday);
```

To see which days belong to each variable, type:

```
teamMtgs  
allMtgs  
  
teamMtgs =  
Wednesday, Friday  
  
allMtgs =  
Monday, Wednesday, Friday
```

Removing a Flag from a Variable

Suppose that your manager cancels the Wednesday meeting this week. To remove Wednesday from the `allMtgs` variable, use the `bitxor` method.

```
thisWeekMtgs = bitxor(allMtgs,NetDocEnum.MyDays.Wednesday)  
  
thisWeekMtgs =  
Monday, Friday
```

Using a bitwise method such as `bitxor` on `allMtgs` does not modify the value of `allMtgs`. This example creates a variable, `thisWeekMtgs`, which contains the result of the operation.

Replacing a Flag in a Variable

Suppose that you change the team meeting permanently from Wednesday to Thursday. To remove Wednesday, use `bitxor`, and use `bitor` to add Thursday. Since this is a permanent change, update the `teamMtgs` and `allMtgs` variables.

```
teamMtgs = bitor(...
    (bitand(teamMtgs,...
        bitnot(NetDocEnum.MyDays.Wednesday)),...
    NetDocEnum.MyDays.Thursday);
allMtgs = bitor(teamMtgs,...
    NetDocEnum.MyDays.Monday);
teamMtgs
allMtgs

teamMtgs =
Thursday, Friday

allMtgs =
Monday, Thursday, Friday
```

Testing for Membership

Create the following `RemindMe` function:

```
function RemindMe(day)
% day = NetDocEnum.MyDays enumeration
teamMtgs = bitor(...
    NetDocEnum.MyDays.Friday,...
    NetDocEnum.MyDays.Wednesday);
allMtgs = bitor(teamMtgs,...
    NetDocEnum.MyDays.Monday);

if eq(day,bitand(day,teamMtgs))
    disp('Team meeting today.')
elseif eq(day,bitand(day,allMtgs))
    disp('Meeting today.')
else
    disp('No meetings today!')
end
end
```

Use the `RemindMe` function:


```
today = NetDocEnum.MyDays.Monday;  
RemindMe(today)
```

```
Meeting today.
```

See Also

[bitand](#) | [bitnot](#) | [bitor](#) | [bitxor](#)

Related Examples

- [“NetDocEnum Example Assembly”](#) on page 15-118

Read Special System Folder Path

```
function result = getSpecialFolder(arg)
% Returns the special system folders such as "Desktop", "MyMusic" etc.
% arg can be any one of the enum element mentioned in this link
% http://msdn.microsoft.com/en-us/library/
% system.environment.specialfolder.aspx
% e.g.
%     >> getSpecialFolder('Desktop')
%
%     ans =
%     C:\Users\jsmith\Desktop

% Get the type of SpecialFolder enum, this is a nested enum type.
specialFolderType = System.Type.GetType(...
    'System.Environment+SpecialFolder');
% Get a list of all SpecialFolder enum values
folders = System.Enum.GetValues(specialFolderType);
enumArg = [];

% Find the matching enum value requested by the user
for i = 1:folders.Length
    if (strcmp(char(folders(i)), arg))
        enumArg = folders(i);
        break
    end
end

% Validate
if isempty(enumArg)
    error('Invalid Argument')
end

% Call GetFolderPath method and return the result
result = System.Environment.GetFolderPath(enumArg);
end
```

See Also

Related Examples

- “Iterate Through a .NET Enumeration” on page 15-125

Default Methods for an Enumeration

By default, MATLAB provides the following methods for a .NET enumeration:

- Relational operators — `eq`, `ne`, `ge`, `gt`, `le`, and `lt`.
- Conversion methods — `char`, `double`, and a method to get the underlying value.
- Bitwise methods — Only for enumerations with the `System.Flags` attribute.

For example, type:

```
methods('System.DayOfWeek')
```

```
Methods for class System.DayOfWeek:
```

```
CompareTo    eq
DayOfWeek    ge
Equals        gt
GetHashCode  int32
GetType      le
GetTypeCode  lt
ToString     ne
char
double
```

The method to get the underlying value is `int32`.

The `NetDocEnum.MyDays` enumeration, which has the `Flags` attribute, has the bitwise methods. To list the methods, type:

```
methods('NetDocEnum.MyDays')
```

```
Methods for class NetDocEnum.MyDays:
```

```
CompareTo    char
Equals        double
GetHashCode  eq
GetType      ge
GetTypeCode  gt
MyDays       int32
ToString     le
bitand       lt
bitnot       ne
bitor
bitxor
```

See Also

Related Examples

- “Using Relational Operations” on page 15-127
- “Using Switch Statements” on page 15-127
- “Display .NET Enumeration Members as Character Vectors” on page 15-123
- “NetDocEnum Example Assembly” on page 15-118

More About

- “Use Bit Flags with .NET Enumerations” on page 15-110

NetDocEnum Example Assembly

The C# example `NetDocEnum.cs`, in the `matlabroot/extern/examples/NET/NetSample` folder, defines enumerations used in examples. To see the code, open the file in MATLAB Editor. To run the examples, build the `NetDocEnum` assembly as described in “Build a .NET Application for MATLAB Examples” on page 15-24.

If the `NetDocEnum` assembly is in your `c:\work` folder, load the file:

```
dllPath = fullfile('c:', 'work', 'NetDocEnum.dll');
asm = NET.addAssembly(dllPath);
asm.Enums

ans =
    'NetDocEnum.MyDays'
    'NetDocEnum.Range'
```

See Also

More About

- “Refer to a .NET Enumeration Member” on page 15-121
- “Use Bit Flags with .NET Enumerations” on page 15-110
- “Underlying Enumeration Values” on page 15-129

Work with Members of a .NET Enumeration

To display the member names of an enumeration, use the MATLAB enumeration function. For example, to list the member names of the `System.DayOfWeek` enumeration, type:

```
enumeration('System.DayOfWeek')

Enumeration members for class 'System.DayOfWeek':
    Sunday
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
```

You cannot use the enumeration command to return arrays of .NET enumeration objects. You can read the names and values of the enumeration into arrays, using the `System.Enum` methods `GetNames`, `GetValues`, and `GetType`.

For example, to create arrays `allNames` and `allValues` for the `System.DayOfWeek` enumeration, type:

```
myDay = System.DayOfWeek;
allNames = System.Enum.GetNames(myDay.GetType);
allValues = System.Enum.GetValues(myDay.GetType);
```

The class of the names array is `System.String`, while the class of the values array is the enumeration type `System.DayOfWeek`.

```
whos all*

   Name      Size  Bytes  Class
   allNames  1x1   112   System.String[]
   allValues  1x1   112   System.DayOfWeek[]
```

Although the types are different, the information MATLAB displays is the same. For example, type:

```
allNames(1)

ans =
Sunday
```

Type:

```
allValues(1)
```

```
ans =  
Sunday
```

See Also

enumeration

Related Examples

- “Iterate Through a .NET Enumeration” on page 15-125

More About

- “Information About System.Enum Methods” on page 15-125
- “How MATLAB Handles System.String” on page 15-59

Refer to a .NET Enumeration Member

You use an enumeration member in your code as an instance of an enumeration. To refer to an enumeration member, use the C# namespace, enumeration, and member names:

Namespace.EnumName.MemberName

For example, the `System` namespace in the .NET Framework class library has a `DayOfWeek` enumeration. The members of this enumeration are:

Enumeration members for class 'System.DayOfWeek':

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

To create a variable with the value `Thursday`, type:

```
gameDay = System.DayOfWeek.Thursday;
whos
```

Name	Size	Bytes	Class
gameDay	1x1	104	System.DayOfWeek

Using the Implicit Constructor

The implicit constructor, *Namespace.EnumName*, creates a member with the default value of the underlying type. For example, the `NetDocEnum.Range` enumeration has the following members:

Enumeration members for class 'NetDocEnum.Range':

```
Max
Min
```

Type:

```
x = NetDocEnum.Range
whos x
```

```
x =
0
```

Name	Size	Bytes	Class
x	1x1	104	NetDocEnum.Range

See Also

Related Examples

- “NetDocEnum Example Assembly” on page 15-118

Display .NET Enumeration Members as Character Vectors

To get the descriptive name of an enumeration, use the `char` method. For example, type:

```
gameDay = System.DayOfWeek.Thursday;  
['Next volleyball game is ',char(gameDay)]
```

```
ans =  
Next volleyball game is Thursday
```

Convert .NET Enumeration Values to Type Double

To convert a value to a MATLAB double, type:

```
gameDay = System.DayOfWeek.Thursday;  
myValue = double(gameDay)
```

```
myValue =  
    4
```

Iterate Through a .NET Enumeration

In this section...

“Information About System.Enum Methods” on page 15-125

“Display Enumeration Member Names” on page 15-126

Information About System.Enum Methods

To create MATLAB arrays from an enumeration, use the static `System.Enum` methods `GetNames` and `GetValues`. The input argument for these methods is an enumeration type. Use the `GetType` method for the type of the current instance. To display the signatures for these methods, type:

```
methodsview('System.Enum')
```

Look at the following signatures:

Qualifiers	Return Type	Name	Arguments
	<code>System.Type</code>	<code>GetType</code>	<code>(System.Enum this)</code>
Static	<code>System.String[]</code>	<code>GetNames</code>	<code>(System.Type enumType)</code>
Static	<code>System.Array</code>	<code>GetValues</code>	<code>(System.Type enumType)</code>

To use `GetType`, create an instance of the enumeration. For example:

```
myEnum = System.DayOfWeek;
```

The `enumType` for `myEnum` is:

```
myEnumType = myEnum.GetType;
```

To create an array of names using the `GetNames` method, type:

```
allNames = System.Enum.GetNames(myEnumType);
```

Alternatively:

```
allNames = System.Enum.GetNames(myEnum.GetType);
```

Display Enumeration Member Names

To display all member names of the `System.DayOfWeek` enumeration, create a `System.String` array of names. Use the `Length` property of this array to find the number of members. For example:

```
myDay = System.DayOfWeek;  
allNames = System.Enum.GetNames(myDay.GetType);  
disp(['Members of ' class(myDay)])  
for idx = 1:allNames.Length  
    disp(allNames(idx))  
end
```

```
Members of System.DayOfWeek  
Sunday  
Monday  
Tuesday  
Wednesday  
Thursday  
Friday  
Saturday
```

See Also

Related Examples

- “Read Special System Folder Path” on page 15-114

Use .NET Enumerations to Test for Conditions

In this section...

“Using Switch Statements” on page 15-127

“Using Relational Operations” on page 15-127

With relational operators, you can use enumeration members in `if` and `switch` statements and other functions that test for equality.

Using Switch Statements

The following `Reminder` function displays a message depending on the day of the week:

```
function Reminder(day)
% day = System.DayOfWeek enumeration value
% Add error checking here
switch(day)
    case System.DayOfWeek.Monday
        disp('Department meeting at 10:00')
    case System.DayOfWeek.Tuesday
        disp('Meeting Free Day!')
    case {System.DayOfWeek.Wednesday System.DayOfWeek.Friday}
        disp('Team meeting at 2:00')
    case System.DayOfWeek.Thursday
        disp('Volley ball night')
end
end
```

For example, type:

```
today = System.DayOfWeek.Wednesday;
Reminder(today)

ans =
Team meeting at 2:00
```

Using Relational Operations

Create the following function to display a message:

```
function VolleyballMessage(day)
% day = System.DayOfWeek enumeration value
```

```
if gt(day, System.DayOfWeek.Thursday)
    disp('See you next week at volleyball.')
else
    disp('See you Thursday!')
end
end
```

For a day before Thursday:

```
myDay = System.DayOfWeek.Monday;
VolleyballMessage(myDay)
```

See you Thursday!

For a day after Thursday:

```
myDay = System.DayOfWeek.Friday;
VolleyballMessage(myDay)
```

See you next week at volleyball.

Underlying Enumeration Values

MATLAB supports enumerations of any numeric type.

To find the underlying type of an enumeration, use the `System.Enum` static method `GetUnderlyingType`. For example, the following C# statement in the `NetDocEnum` assembly declares the enumeration `Range`:

```
public enum Range : long {Max = 2147483648L,Min = 255L}
```

To display the underlying type:

```
maxValue = NetDocEnum.Range.Max;  
System.Enum.GetUnderlyingType(maxValue.GetType).FullName  
  
ans =  
System.Int64
```

See Also

Related Examples

- “NetDocEnum Example Assembly” on page 15-118

Limitations to Support of .NET Enumerations

You cannot create arrays of .NET enumerations, or any .NET objects, in MATLAB.

Create .NET Collections

This example uses two `System.String` arrays, `d1` and `d2`, to create a generic collection list. It shows how to manipulate the list and access its members. To create the arrays, type:

```
d1 = NET.createArray('System.String',3);
d1(1) = 'Brachiosaurus';
d1(2) = 'Shunosaurus';
d1(3) = 'Allosaurus';

d2 = NET.createArray('System.String',4);
d2(1) = 'Tyrannosaurus';
d2(2) = 'Spinosaurus';
d2(3) = 'Velociraptor';
d2(4) = 'Triceratops';
```

Create a generic collection, `dc`, to contain `d1`. The `System.Collections.Generic.List` class is in the `microsoft.collections.generic` assembly, which MATLAB loads automatically.

```
dc = NET.createGeneric('System.Collections.Generic.List',{'System.String'},3)

List<System*String> handle

    Capacity: 3
    Count: 0
```

The `List` object `dc` has a `Capacity` of three, but currently is empty (`Count = 0`).

Use the `AddRange` method to add the contents of `d1` to the list. For more information, search the Web for `System.Collections.Generic` and select the `List` class.

```
AddRange(dc,d1);
```

List `dc` now has three items:

```
dc.Count
```

To display the contents, use the `Item` method and zero-based indexing:

```
for i = 1:dc.Count
    disp(dc.Item(i-1))
end
```

```
Brachiosaurus  
Shunosaurus  
Allosaurus
```

Another way to add values is to use the `InsertRange` method. Insert the `d2` array starting at index 1:

```
InsertRange(dc,1,d2);
```

The size of the array has grown to seven. To display the values, type:

```
for i = 1:dc.Count  
    disp(dc.Item(i-1))  
end
```

```
Brachiosaurus  
Tyrannosaurus  
Spinosaurus  
Velociraptor  
Triceratops  
Shunosaurus  
Allosaurus
```

The first item in the `d2` array ('Tyrannosaurus') is at index 1 in list `dc`:

```
System.String.Compare(d2(1),dc.Item(1))
```

The `System.String.Compare` answer, 0, indicates that the two values are equal.

See Also

Related Examples

- “Convert .NET Collections to MATLAB Arrays” on page 15-133

External Websites

- .NET Framework 4.5 `System.Collections.Generic.List` Class

Convert .NET Collections to MATLAB Arrays

Use the `ToArray` method of the `System.Collections.Generic.List` class to convert a collection to an array. For example, use `GetRange` to get three values from a list. Then call `ToArray` to create a `System.String` array.

```
dog = NET.createArray('System.String',3);
dog(1) = 'poodle';
dog(2) = 'spaniel';
dog(3) = 'Irish setter';
dc = NET.createGeneric('System.Collections.Generic.List',{'System.String'},3);
AddRange(dc,dog);
temp = GetRange(dc,0,3);
dArr = ToArray(temp);
```

Create a MATLAB array `Dogs`:

```
Dogs = {char(dArr(1)),char(dArr(2)),char(dArr(3))}
```

```
Dogs =
    'poodle'    'spaniel'    'Irish setter'
```

Now you can use `Dogs` in MATLAB functions. For example, sort the array alphabetically:

```
sort(Dogs)'
```

```
ans =
    'Irish setter'
    'poodle'
    'spaniel'
```

See Also

Related Examples

- “Create .NET Collections” on page 15-131

Create .NET Arrays of Generic Type

This example creates a .NET array of `List<Int32>` generic type.

```
genType = NET.GenericClass('System.Collections.Generic.List', ...  
    'System.Int32');  
arr = NET.createArray(genType, 5)
```

```
arr =
```

```
List<System*Int32>[] with properties:  
    Length: 5  
    LongLength: 5  
    Rank: 1  
    SyncRoot: [1x1 System.Collections.Generic.List<System*Int32>[]]  
    IsReadOnly: 0  
    IsFixedSize: 1  
    IsSynchronized: 0
```

Display .NET Generic Methods Using Reflection

In this section...

“showGenericMethods Function” on page 15-135

“Display Generic Methods in a Class” on page 15-136

“Display Generic Methods in a Generic Class” on page 15-137

showGenericMethods Function

The showGenericMethods function reads a .NET object or a fully qualified class name and returns a cell array of the names of the generic method in the given class or object. Create the following MATLAB functions:

```
function output = showGenericMethods(input)
% if input is a .NET object, get MethodInfo[]
if IsNetObject(input)
    methods = GetType.GetMethods(input);
    % if input is a string, get the type and get get MethodInfo[]
elseif ischar(input) && ~isempty(input)
    type = getType(input);
    if isempty(type)
        disp(strcat(input, ' not found'))
        return
    end
    methods = GetMethods(type);
else
    return
end
% generate generic method names from MethodInfo[]
output = populateGenericMethods(methods);

end

function output = populateGenericMethods(methods)
% generate generic method names from MethodInfo[]
index = 1;
for i = 1:methods.Length
    method = methods(i);
    if method.IsGenericMethod
        output{index,1} = method.ToString.char;
        index = index + 1;
    end
end
end
```

```
    end
end
end

function result = IsNetObject(input)
% Must be sub class of System.Object to be a .NET object
result = isa(input, 'System.Object');
end

function outputType = getType(input)
% Input is a string representing the class name
% First try the static GetType method of Type handle.
% This method can find any type from
% System or mscorlib assemblies
outputType = System.Type.GetType(input, false, false);
if isempty(outputType)
    % Framework's method to get the type failed.
    % Manually look for it in
    % each assembly visible to MATLAB
    assemblies = System.AppDomain.CurrentDomain.GetAssemblies;
    for i = 1:assemblies.Length
        asm = assemblies.Get(i-1);
        % Look for a particular type in the assembly
        outputType = GetType(asm, input, false, false);
        if ~isempty(outputType)
            % Found the type - done
            break
        end
    end
end
end
end
```

Display Generic Methods in a Class

The NetDocGeneric assembly contains a class with generic methods.

```
dllPath = fullfile('c:', 'work', 'NetDocGeneric.dll');
asm = NET.addAssembly(dllPath);
asm.Classes

ans =
    'NetDocGeneric.SampleClass'
```

Display the methods in SampleClass:


```
showGenericMethods('NetDocGeneric.SampleClass')
ans =
'K GenMethod[K](K)'
'K GenMethodWithMixedArgs[K](K, K, Boolean)'
'K GenStaticMethod[K](K)'
'K GenStaticMethodWithMixedArgs[K](K, K, Boolean)'
```

Display Generic Methods in a Generic Class

The `NetDocGeneric` assembly contains a generic class with generic methods.

```
dllPath = fullfile('c:', 'work', 'NetDocGeneric.dll');
asm = NET.addAssembly(dllPath);
asm.GenericTypes
```

```
ans =
'NetDocGeneric.SampleGenericClass`1[T]'
```

Display the methods in `SampleGenericClass`:

```
cls = NET.createGeneric('NetDocGeneric.SampleGenericClass', {'System.Double'});
showGenericMethods(cls)
```

```
ans =
'System.String ParameterizedGenMethod[K](Double, K)'
'T GenMethod[T](T)'
'K GenStaticMethod[K](K)'
'K GenStaticMethodWithMixedArgs[K](K, K, Boolean)'
'System.String ParameterizedStaticGenMethod[K](Double, K)'
```

See Also

More About

- “Call .NET Generic Methods” on page 15-140

.NET Generic Classes

Generics are classes and methods that have placeholders (type parameters or parameterized types) for one or more types. This lets you design classes that take in a generic type and determine the actual type at run time. A common use for generic classes is to work with collections. For information about generic methods, see “Call .NET Generic Methods” on page 15-140.

The `NET.createGeneric` function creates an instance of the specialized generic class given the following:

- Fully qualified name of the generic class definition
- List of fully qualified parameter type names for generic type specialization
- Variable list of constructor arguments

Use instances of the `NET.GenericClass` helper class in `NET.createGeneric` function’s parameter type list when specialization requires another parameterized class definition. The class instances serve as parameterized data type definitions and are constructed using fully qualified generic type name and a variable length list of fully qualified type names for generic type specialization. This list can also contain instances of `NET.GenericClass` if an extra nested level of parameterization is required.

Accessing Items in .NET Collections

Use the `Item` property of the `System.Collections.Generic.List` class to get or set an element at a specified index. Since `Item` is a property that takes arguments, MATLAB maps it to a pair of methods to get and set the value. For example, the syntax to use `Item` to get a value is:

Return Type	Name	Arguments
<code>System.String</code> <code>RetVal</code>	<code>Item</code>	<code>(System.Collections.Generic.List<System*String> this, int32 scalar index)</code>

The syntax to use `Item` to set a value is:

Return Type	Name	Arguments
<code>none</code>	<code>Item</code>	<code>(System.Collections.Generic.List<System*String> this, int32 scalar index, System.String value)</code>

See Also

Related Examples

- “Create .NET Collections” on page 15-131

Call .NET Generic Methods

A generic method declares one or more parameterized types. For more information, search for the term `generics` in the .NET Framework Class Library, as described in “To Learn More About the .NET Framework” on page 15-21.

Use the `NET.invokeGenericMethod` function to call a generic method. How you use the `NET.invokeGenericMethod` depends if the method is static or if it is a member of a generic class.

Using the NetDocGeneric Example

The C# example `NetDocGeneric.cs`, in the `matlabroot/extern/examples/NET/NetSample` folder, defines simple generic methods to illustrate the `NET.invokeGenericMethod` syntax. To see the code, open the file in MATLAB Editor. Build the `NetDocGeneric` assembly as described in “Build a .NET Application for MATLAB Examples” on page 15-24.

If you created the assembly `NetDocGeneric` and put it in your `c:\work` folder, type the following MATLAB commands to load the assembly:

```
dllPath = fullfile('c:', 'work', 'NetDocGeneric.dll');  
NET.addAssembly(dllPath);
```

Note The `methods` and `methodsview` functions do not list generic methods. Use the “Display .NET Generic Methods Using Reflection” on page 15-135 example.

Invoke Generic Class Member Function

The `GenMethod` method in `NetDocGeneric.SampleClass` returns the input argument as type `K`. To call `GenMethod`, create an object, `cls`:

```
cls = NetDocGeneric.SampleClass();
```

To convert 5 to an integer parameter type, such as `System.Int32`, call `NET.invokeGenericMethod` with the object:

```
ret = NET.invokeGenericMethod(cls, ...  
    'GenMethod', ...
```

```
{'System.Int32'}, ...
5);
```

The `GenMethodWithMixedArgs` method has parameterized typed arguments, `arg1` and `arg2`, and a strongly-typed argument, `tf`, of type `bool`. The `tf` flag controls which argument `GenMethodWithMixedArgs` returns. To return `arg1`, use the syntax:

```
ret = NET.invokeGenericMethod(cls, 'GenMethodWithMixedArgs', ...
    {'System.Double'}, 5, 6, true);
```

To return `arg2`, use the syntax:

```
ret = NET.invokeGenericMethod(cls, 'GenMethodWithMixedArgs', ...
    {'System.Double'}, 5, 6, false);
```

Invoke Static Generic Functions

To invoke static method `GenStaticMethod`, call `NET.invokeGenericMethod` with the fully qualified class name:

```
ret = NET.invokeGenericMethod('NetDocGeneric.SampleClass', ...
    'GenStaticMethod', ...
    {'System.Int32'}, ...
    5);
```

Invoke Static Generic Functions of a Generic Class

If a static function is a member of a generic class, create a class definition using the `NET.GenericClass` constructor:

```
genClsDef = NET.GenericClass('NetDocGeneric.SampleGenericClass', ...
    'System.Double');
```

To invoke static method `GenStaticMethod` of `SampleGenericClass`, call `NET.invokeGenericMethod` with the class definition:

```
ret = NET.invokeGenericMethod(genClsDef, ...
    'GenStaticMethod', ...
    {'System.Int32'}, ...
    5);
```

Invoke Generic Functions of a Generic Class

If a generic method uses the same parameterized type as the generic class, you can call the function directly on the class object. If the generic uses a different type than the class, use the `NET.invokeGenericMethod` function.

See Also

More About

- “Display .NET Generic Methods Using Reflection” on page 15-135

Using COM Objects from MATLAB

- “MATLAB COM Integration” on page 16-2
- “Register Servers” on page 16-4
- “Get Started with COM” on page 16-5
- “Read Spreadsheet Data Using Excel as Automation Server” on page 16-6
- “Supported Client/Server Configurations” on page 16-13

MATLAB COM Integration

Concepts and Terminology

While the ideas behind COM technology are straightforward, the terminology is not. The meaning of COM terms has changed over time and few concise definitions exist. Here are some terms that you should be familiar with. These are not comprehensive definitions. For a complete description of COM, you'll need to consult outside resources.

COM Objects, Clients, and Servers

A COM object is a software component that conforms to the Component Object Model. COM enforces encapsulation of the object, preventing direct access of its data and implementation. COM objects expose interfaces, which consist of properties, methods and events.

A COM client is a program that makes use of COM objects. COM objects that expose functionality for use are called COM servers. COM servers can be in-process or out-of-process. An example of an out-of-process server is Microsoft Excel spreadsheet program.

MATLAB can be used as either a COM client or a COM Automation server.

Interfaces

The functionality of a component is defined by one or more interfaces. To use a COM component, you must learn about its interfaces, and the methods, properties, and events implemented by the component. The component vendor provides this information.

There are two standard COM interfaces:

- **IUnknown** — An interface required by all COM components. All other COM interfaces are derived from IUnknown.
- **IDispatch** — An interface that exposes objects, methods and properties to applications that support Automation.

MATLAB COM Client

A COM client is a program that manipulates COM objects. These objects can run in the MATLAB application or can be part of another application that exposes its objects as a programmatic interface to the application.

Using MATLAB as a COM client provides two techniques for developing programs in MATLAB:

- You can include COM components in your MATLAB application (for example, a spreadsheet).
- You can access existing applications that expose objects via Automation.

MATLAB COM clients can access applications that support Automation, such as the Excel spreadsheet program. MATLAB creates an Automation server in which to run the application and returns a handle to the primary interface for the object created.

MATLAB COM Automation Server

Automation provides an infrastructure whereby applications called automation controllers can access and manipulate (i.e. set properties of or call methods on) shared automation objects that are exported by other applications, called Automation servers. Any Windows program that can be configured as an Automation controller can control MATLAB.

For example, using Microsoft Visual Basic® programming language, you can run a MATLAB script in a Microsoft PowerPoint® presentation. In this case, PowerPoint is the controller and MATLAB is the server.

See Also

More About

- “Create COM Objects” on page 17-2
- “Calling MATLAB as COM Automation Server”

Register Servers

Before using COM objects, you must register their servers. Most are registered by default. However, if you get a new `.ocx`, `.dll`, or other object file for the server, you must register the file manually in the Windows registry.

Use the Windows `regsvr32` command to register your file. From the Windows prompt, use the `cd` function to go to the folder containing the object file. If your object file is an `.ocx` file, type:

```
regsvr32 filename.ocx
```

If you encounter problems with this procedure, consult a Windows manual or contact your local system administrator.

Access COM Controls Created with .NET

If you create a COM control using Microsoft .NET Framework 4, use the DOS `regasm` command with the `/codebase` option to register your file.

Verify Registration

To verify that a server is registered, refer to your Microsoft product documentation for information about using Microsoft Visual Studio or the Microsoft Registry Editor programs.

Get Started with COM

Create Instance of COM Object

Use the `actxserver` function to create and manipulate objects from MATLAB that are exposed in an application that supports Automation. The function returns a handle to the object's main interface, which you use to access the object's methods, properties, and events, and any other interfaces it provides.

Register Custom Control

If your MATLAB program uses a custom control, for example, one that you have created especially for your application, you must register it with the Microsoft Windows operating system before you can use it. You can do this from your MATLAB program by issuing an operating system command:

```
!regsvr32 /s filename.ocx
```

where *filename* is the name of the file containing the control. Using this command in your program enables you to provide custom-made controls that you make available to other users by registering the control on their computer when they run your MATLAB program.

See Also

`actxserver` | `events` | `get` | `methods` | `set`

More About

- “Register Servers” on page 16-4
- “COM Methods” on page 17-15
- “COM Events” on page 17-18
- “COM Object Interfaces” on page 17-22

Read Spreadsheet Data Using Excel as Automation Server

This example shows how to use a COM Automation server to access another application from MATLAB. It creates a user interface to access the data in a Microsoft Excel file. If you do not use the Component Object Model (COM) in your applications, then see the functions and examples in “Spreadsheets” for alternatives to importing Excel spreadsheet data into MATLAB.

To enable the communication between MATLAB and the spreadsheet program, this example creates an object in an Automation server running an Excel application. MATLAB then accesses the data in the spreadsheet through the interfaces provided by the Excel Automation server. Finally, the example creates a user interface to access the data in a Microsoft Excel file.

Techniques Demonstrated

- Use of an Automation server to access another application from MATLAB
- Ways to manipulate Excel data into types used in the interface and plotting

The following techniques demonstrate how to visualize and manipulate the spreadsheet data:

- Implementation of an interface that enables plotting of selected columns of the Excel spreadsheet.
- Insertion of a MATLAB figure into an Excel file.

To view the complete code listing, open the file `actx_excel.m` in the editor.

Create Excel Automation Server

The first step in accessing the spreadsheet data from MATLAB is to run the Excel application in an Automation server process using the `actxserver` function and the program ID, `excel.application`.

```
exl = actxserver('excel.application');
```

The `exl` object provides access to a number of interfaces supported by the Excel program. Use the `Workbooks` interface to open the Excel file containing the data.

```
exlWkbk = exl.Workbooks;
exlFile = exlWkbk.Open([docroot '/techdoc/matlab_external/examples/input_resp_data.xls']);
```

Use the workbook `Sheets` interface to access the data from a `Range` object, which stores a reference to a range of data from the specified sheet. This example accesses all the data from the first cell in column A to the last cell in column G.

```
exlSheet1 = exlFile.Sheets.Item('Sheet1');
robj = exlSheet1.Columns.End(4); % Find the end of the column
numrows = robj.row; % And determine what row it is
dat_range = ['A1:G' num2str(numrows)]; % Read to the last row
rngObj = exlSheet1.Range(dat_range);
```

At this point, the entire data set from the Excel file's `sheet1` is accessed via the range object interface `rngObj`. This object returns the data in a MATLAB cell array `exlData`, which contains both numeric and character data:

```
exlData = rngObj.Value;
```

Manipulate Data in MATLAB Workspace

Now that the data is in a cell array, you can use MATLAB functions to extract and reshape parts of the data to use in the interface and to pass to the plot function. For assumptions about the data, see “Excel Spreadsheet Format” on page 16-8.

The following code manipulates the data:

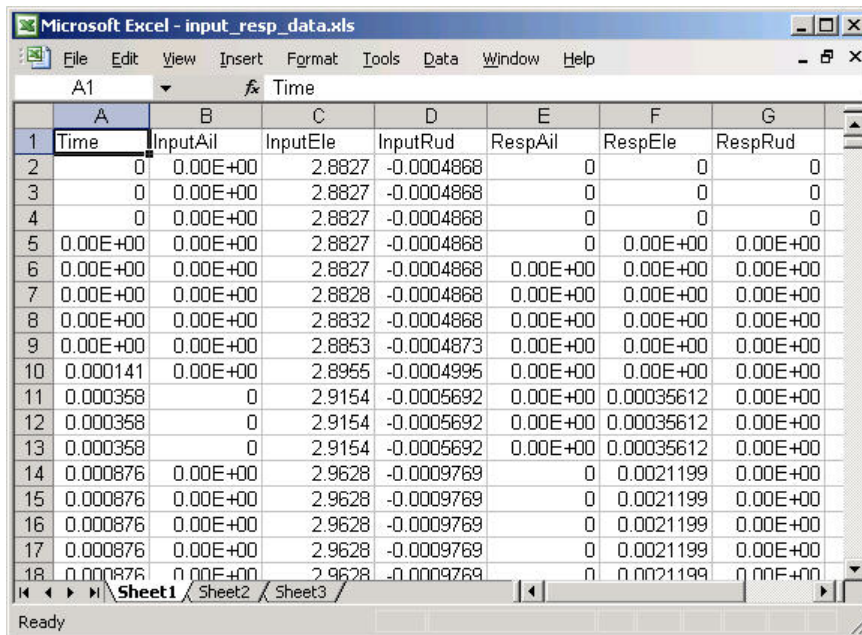
```
for ii = 1:size(exlData,2)
    matData(:,ii) = reshape([exlData{2:end,ii}],size(exlData(2:end,ii)));
    listBox{ii} = [exlData{1,ii}];
end
```

The code performs the following operations:

- Extracts numeric data from the cell array. See the indexing expression inside the curly braces `{}`.
- Concatenates the individual doubles returned by the indexing operation. See the expression inside the square brackets `[]`.
- Reshapes the results into an array that arranges the data in columns using the `reshape` function.
- Extracts the text in the first cell in each column of `exlData` data and stores the text in a cell array `listBox`. This variable is used to generate the items in the list box.

Excel Spreadsheet Format

This example assumes a particular organization of the Excel spreadsheet, as shown in the following picture.



	A	B	C	D	E	F	G
1	Time	InputAil	InputEle	InputRud	RespAil	RespEle	RespRud
2	0	0.00E+00	2.8827	-0.0004868	0	0	0
3	0	0.00E+00	2.8827	-0.0004868	0	0	0
4	0	0.00E+00	2.8827	-0.0004868	0	0	0
5	0.00E+00	0.00E+00	2.8827	-0.0004868	0	0.00E+00	0.00E+00
6	0.00E+00	0.00E+00	2.8827	-0.0004868	0.00E+00	0.00E+00	0.00E+00
7	0.00E+00	0.00E+00	2.8828	-0.0004868	0.00E+00	0.00E+00	0.00E+00
8	0.00E+00	0.00E+00	2.8832	-0.0004868	0.00E+00	0.00E+00	0.00E+00
9	0.00E+00	0.00E+00	2.8853	-0.0004873	0.00E+00	0.00E+00	0.00E+00
10	0.000141	0.00E+00	2.8955	-0.0004995	0.00E+00	0.00E+00	0.00E+00
11	0.000358	0	2.9154	-0.0005692	0.00E+00	0.00035612	0.00E+00
12	0.000358	0	2.9154	-0.0005692	0.00E+00	0.00035612	0.00E+00
13	0.000358	0	2.9154	-0.0005692	0.00E+00	0.00035612	0.00E+00
14	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
15	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
16	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
17	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
18	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00

The format of the Excel file is as follows:

- The first element in each column is text that identifies the data contained in the column. These values are extracted and used to populate the list box.
- The first column Time is used for the x-axis of all plots of the remaining data.
- All rows in each column are read into MATLAB.

Create Plotter Interface

This example uses an interface that enables you to select from a list of input and response data. All data is plotted as a function of time and you can continue to add more data to the graph. Each data plot added to the graph causes the legend to expand.

The interface includes these details:

- Legend that updates as you add data to a graph
- Clear button that enables you to clear all graphs from the axes
- Save button that saves the graph as a PNG file and adds it to another Excel file
- Toggle button that shows or hides the Excel file being accessed
- Figure delete function to terminate the Automation server

Select and Plot Data

When you click the **Create Plot** button, its callback function queries the list box to determine what items are selected and plots each data versus time. MATLAB updates the legend to display new data while still maintaining the legend for the existing data.

```
function plotButtonCallback(src,evnt)
iSelected = get(listBox,'Value');
grid(a,'on');hold all
for p = 1:length(iSelected)
    switch iSelected(p)
        case 1
            plot(a,tme,matData(:,2))
        case 2
            plot(a,tme,matData(:,3))
        case 3
            plot(a,tme,matData(:,4))
        case 4
            plot(a,tme,matData(:,5))
        case 5
            plot(a,tme,matData(:,6))
        case 6
            plot(a,tme,matData(:,7))
        otherwise
            disp('Select data to plot')
    end
end
[b,c,g,lbs] = legend([lbs listBoxList(iSelected+1)]);
end % plotButtonCallback
```

Clear the Axes

The plotter is designed to continually add graphs as the user selects data from the list box. The **Clear Graph** button clears and resets the axes and clears the variable used to store the labels of the plot data (used by legend).

```
%% Callback for clear button
function clearButtonCallback(src,evt)
    cla(a,'reset')
    lbs = '';
end % clearButtonCallback
```

Display or Hide Excel File

The MATLAB program has access to the properties of the Excel application running in the Automation server. By setting the `Visible` property to 1 or 0, this callback controls the visibility of the Excel file.

```
%% Display or hide Excel file
function dispButtonCallback(src,evt)
    exl.visible = get(src,'Value');
end % dispButtonCallback
```

Close Figure and Terminate Excel Automation Process

Since the Excel Automation server runs in a separate process from MATLAB, you must terminate this process explicitly. There is no reason to keep this process running after closing the interface, so this example uses the figure's `delete` function to terminate the Excel process with the `Quit` method. You also need to terminate the Excel process used for saving the graph. For information about terminating this process, see “Insert MATLAB Graphs into Excel Spreadsheet” on page 16-10.

```
%% Terminate Excel processes
function deleteFig(src,evt)
    exlWkbk.Close
    exlWkbk2.Close
    exl.Quit
    exl2.Quit
end % deleteFig
```

Insert MATLAB Graphs into Excel Spreadsheet

You can save the graph created with this interface in an Excel file. This example uses a separate Excel Automation server process for this purpose. The callback for the **Save Graph** push button creates the image and adds it to an Excel file:

- Both the axes and legend are copied to an invisible figure configured to print the graph as you see it on the screen (figure `PaperPositionMode` property is set to `auto`).

- The `print` command creates the PNG image.
- Use the `Shapes` interface to insert the image in the Excel workbook.

The server and interfaces are instanced during the initialization phase:

```
exl2 = actxserver('excel.application');
exlWkbk2 = exl2.Workbooks;
wb = invoke(exlWkbk2, 'Add');
graphSheet = invoke(wb.Sheets, 'Add');
Shapes = graphSheet.Shapes;
```

Use this code to implement the **Save Graph** button callback:

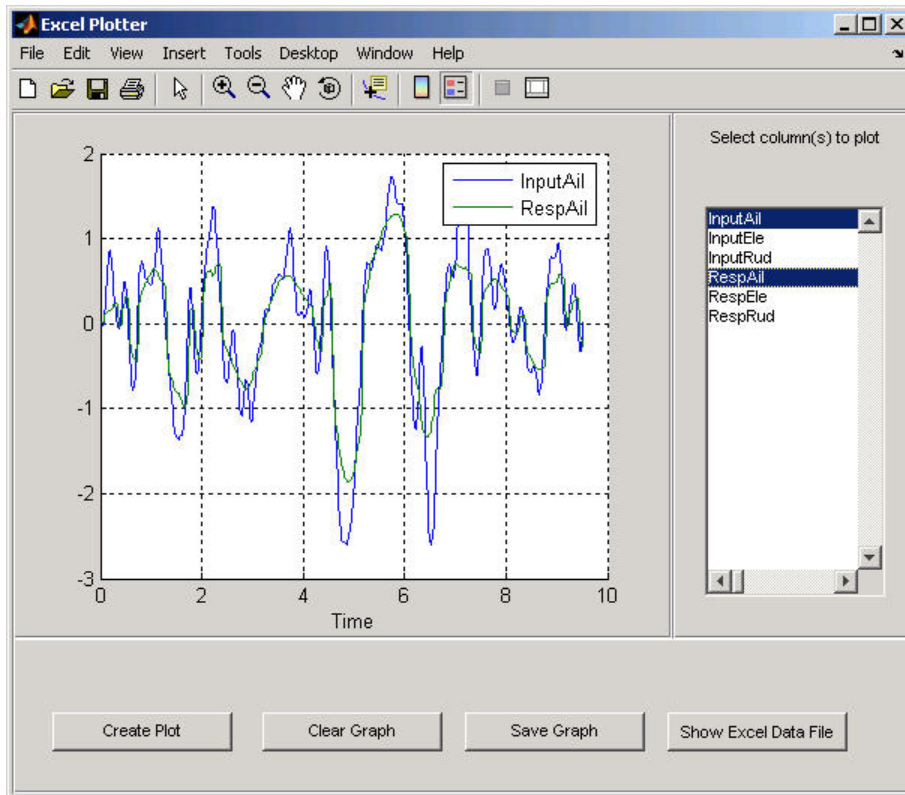
```
function saveButtonCallback(src,evt)
    tempfig = figure('Visible','off','PaperPositionMode','auto');
    tempfigfile = [tempname '.png'];
    ah = findobj(f,'type','axes');
    copyobj(ah,tempfig) % Copy both graph axes and legend axes
    print(tempfig, '-dpng', tempfigfile);
    Shapes.AddPicture(tempfigfile,0,1,50,18,300,235);
    exl2.visible = 1;
end
```

Run Example

To run the example, select any items in the list box and click the **Create Plot** button. The sample data provided with this example contain three input and three associated response data sets. All of these data sets are plotted versus the first column in the Excel file, which is the time data.

View the Excel data file by clicking the **Show Excel Data File** button. To save an image of the graph in a different Excel file, click the **Save Graph** button. If you have write-access permission in the current folder, then the **Save Graph** option creates a temporary PNG file in that folder.

The following image shows the interface with an input/response pair selected in the list box and plotted in the axes.



To run this example, click this link.

See Also

`xlsread`

More About

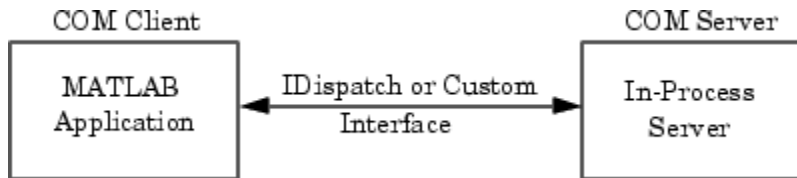
- “Import Spreadsheets”

Supported Client/Server Configurations

You can configure MATLAB to either control or be controlled by other COM components. When MATLAB controls another component, MATLAB is the client, and the other component is the server. When another component controls MATLAB, MATLAB is the server.

MATLAB Client and In-Process Server

The following diagram shows how the MATLAB client interacts with an “In-Process Server” on page 18-5.



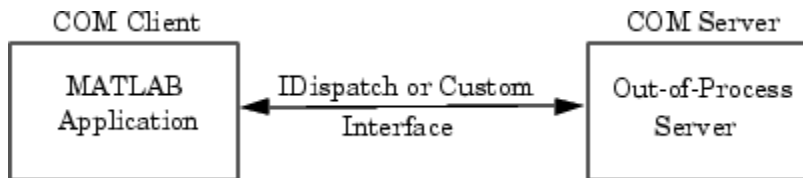
The server exposes its properties and methods through the IDispatch (Automation) interface or a Custom interface, depending on which interfaces the component implements. For information on accessing interfaces, see “COM Object Interfaces” on page 17-22.

Any COM component that has been implemented as a dynamic link library (DLL) is also instantiated in an in-process server. That is, it is created in the same process as the MATLAB client application. When MATLAB uses a DLL server, it runs in a separate window rather than a MATLAB figure window.

To learn more about working with MATLAB as a client, see “Create COM Objects” on page 17-2.

MATLAB Client and Out-of-Process Server

In this configuration, a MATLAB client application interacts with a component that has been implemented as a “Local Out-of-Process Server” on page 18-6. Examples of out-of-process servers are Microsoft Excel and Microsoft Word programs.



As with in-process servers, this server exposes its properties and methods through the IDispatch (Automation) interface or a Custom interface, depending on which interfaces the component implements. For information on accessing interfaces, see “COM Object Interfaces” on page 17-22.

Since the client and server run in separate processes, you have the option of creating the server on any system on the same network as the client.

If the component provides a user interface, its window is separate from the client application.

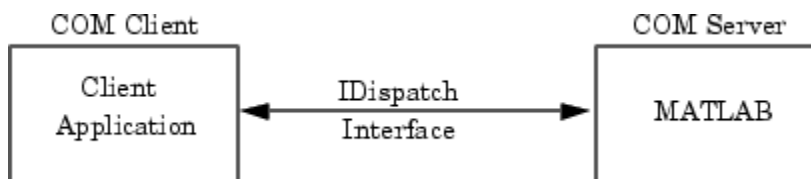
To learn more about working with MATLAB as a client, see “Create COM Objects” on page 17-2.

COM Implementations Supported by MATLAB

MATLAB only supports COM implementations that are compatible with the Microsoft Active Template Library (ATL) API. In general, your COM object should be able to be contained in an ATL host window in order to work with MATLAB.

Client Application and MATLAB Automation Server

MATLAB operates as the Automation server in this configuration. It can be created and controlled by any Microsoft Windows program that can be an *Automation controller*. Some examples of Automation controllers are Microsoft Excel, Microsoft Access™, Microsoft Project, and many Microsoft Visual Basic and Microsoft Visual C++ programs.



MATLAB Automation server capabilities include the ability to execute commands in the MATLAB workspace, and to get and put matrices directly from and into the workspace. You can start a MATLAB server to run in either a shared or dedicated mode. You also have the option of running it on a local or remote system.

To create the MATLAB server from an external application program, use the appropriate function from that language to instantiate the server. (For example, use the Visual Basic `CreateObject` function.) For the programmatic identifier, specify `matlab.application`. To run MATLAB as a dedicated server, use the `matlab.application.single` programmatic identifier. See “Shared and Dedicated Servers” on page 18-5 for more information.

The function that creates the MATLAB server also returns a handle to the properties and methods available in the server through the `IDispatch` interface.

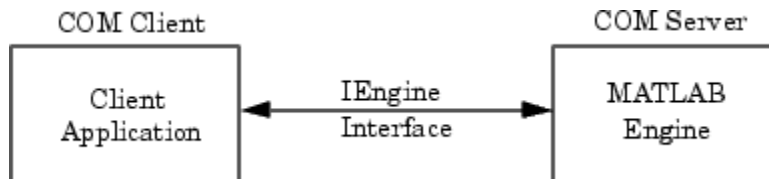
Note Because VBScript client programs require an Automation interface to communicate with servers, this is the only configuration that supports a VBScript client.

For More Information

To learn more about working with Automation servers, see “MATLAB COM Automation Server Interface” on page 18-4.

Client Application and MATLAB Engine Server

MATLAB provides a faster custom interface called `IEngine` for client applications written in C, C++, or Fortran. MATLAB uses `IEngine` to communicate between the client application and the MATLAB engine running as a COM server.



Use the MATLAB Engine API functions to start and end the server process, and to send commands to be processed by MATLAB.

MATLAB COM Client Support

- “Create COM Objects” on page 17-2
- “Handle COM Data in MATLAB” on page 17-4
- “COM Object Properties” on page 17-12
- “COM Methods” on page 17-15
- “COM Events” on page 17-18
- “COM Event Handlers” on page 17-20
- “COM Object Interfaces” on page 17-22
- “Save and Delete COM Objects” on page 17-24
- “MATLAB Application as DCOM Client” on page 17-26
- “Explore COM Objects” on page 17-27
- “Change Row Height in Range of Spreadsheet Cells” on page 17-31
- “Write Data to Excel Spreadsheet” on page 17-34
- “Change Cursor in Spreadsheet” on page 17-36
- “Insert Spreadsheet After First Sheet” on page 17-38
- “Connect to Existing Excel Application” on page 17-40
- “Display Message for Workbook OnClose Event” on page 17-41
- “COM Collections” on page 17-42
- “MATLAB COM Support Limitations” on page 17-43
- “Interpreting Argument Callouts in COM Error Messages” on page 17-44

Create COM Objects

Use the `actxserver` function to create an in-process server for a dynamic link library (DLL) component or an out-of-process server for an executable (EXE) component.

Instantiate DLL Component

To create a server for a component implemented as a dynamic link library (DLL), use the `actxserver` function. MATLAB creates an instance of the component in the same process that contains the client application.

The syntax for `actxserver`, when used with a DLL component, is `actxserver(ProgID)`, where `ProgID` is the programmatic identifier for the component.

`actxserver` returns a handle to the primary interface to the object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain more interfaces to the object. For more information on using interfaces, see “COM Object Interfaces” on page 17-22.

Unlike Microsoft ActiveX controls, any user interface displayed by the server appears in a separate window.

You cannot use a 32-bit in-process DLL COM object in a 64-bit MATLAB application. For information about this restriction, see [Why am I not able to use 32-bit DLL COM Objects in 64-bit MATLAB?](#)

Instantiate EXE Component

You can use the `actxserver` function to create a server for a component implemented as an executable (EXE). In this case, MATLAB instantiates the component in an out-of-process server.

The syntax for `actxserver` to create an executable is `actxserver(ProgID, sysname)`. `ProgID` is the programmatic identifier for the component, and `sysname` is an optional argument used in configuring a distributed COM (DCOM) system.

`actxserver` returns a handle to the primary interface to the COM object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain more interfaces to the object. For more information on using interfaces, see “COM Object Interfaces” on page 17-22.

Any user interface displayed by the server appears in a separate window.

This example creates a COM server application running the Microsoft Excel spreadsheet program. The handle is assigned to `h`.

```
h = actxserver('Excel.Application')
```

MATLAB displays:

```
h =  
    COM.excel.application
```

MATLAB can programmatically connect to an instance of a COM Automation server application that is already running on your computer. To get a reference to such an application, use the `actxGetRunningServer` function.

This example gets a reference to the Excel program, which must already be running on your system. The returned handle is assigned to `h`.

```
h = actxGetRunningServer('Excel.Application')
```

MATLAB displays:

```
h =  
    COM.excel.application
```

See Also

`actxGetRunningServer` | `actxserver`

Handle COM Data in MATLAB

Pass Data to COM Objects

When you use a COM object in a MATLAB command, the MATLAB types you pass in the call are converted to types native to the COM object. MATLAB performs this conversion on each argument that is passed. This section describes the conversion.

MATLAB converts MATLAB arguments into types that best represent the data to the COM object. The following table shows all the MATLAB base types for passed arguments and the COM types defined for input arguments. Each row shows a MATLAB type followed by the possible COM argument matches. For a description of COM variant types, see the table in “Handle Data from COM Objects” on page 17-5.

MATLAB Argument	Closest COM Type	Allowed Types
handle	VT_DISPATCH VT_UNKNOWN	VT_DISPATCH VT_UNKNOWN
character vector	VT_BSTR	VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE
int16	VT_I2	VT_I2
uint16	VT_UI2	VT_UI2
int32	VT_I4	VT_I4 VT_INT
uint32	VT_UI4	VT_UI4 VT_UINT
int64	VT_I8	VT_I8
uint64	VT_UI8	VT_UI8
single	VT_R4	VT_R4

MATLAB Argument	Closest COM Type	Allowed Types
double	VT_R8	VT_R8 VT_CY
logical	VT_BOOL	VT_BOOL
char	VT_I1	VT_I1 VT_UI1

Variant Data

`variant` is any data type except a structure or a sparse array. (For more information, see “Fundamental MATLAB Classes”.)

When used as an input argument, MATLAB treats `variant` and `variant(pointer)` the same way.

If you pass an empty array (`[]`) of type `double`, MATLAB creates a `variant(pointer)` set to `VT_EMPTY`. Passing an empty array of any other numeric type is not supported.

MATLAB Argument	Closest COM Type	Allowed Types
<code>variant</code>	VT_VARIANT	VT_VARIANT VT_USERDEFINED VT_ARRAY
<code>variant(pointer)</code>	VT_VARIANT	VT_VARIANT VT_BYREF

SAFEARRAY Data

When a COM method identifies a `SAFEARRAY` or `SAFEARRAY(pointer)`, the MATLAB equivalent is a matrix.

MATLAB Argument	Closest COM Type	Allowed Types
<code>SAFEARRAY</code>	VT_SAFEARRAY	VT_SAFEARRAY
<code>SAFEARRAY(pointer)</code>	VT_SAFEARRAY	VT_SAFEARRAY VT_BYREF

Handle Data from COM Objects

Data returned from a COM object is often incompatible with MATLAB types. When this occurs, MATLAB converts the returned value to a data type native to the MATLAB

language. This section describes the conversion performed on the various types that can be returned from COM objects.

The following table shows how MATLAB converts data from a COM object into MATLAB variables.

COM Variant Type	Description	MATLAB Representation
VT_DISPATCH	IDispatch *	handle
VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE	wide null terminated string null terminated string OLE Automation string FILETIME SCODE 16-byte fixed point Class ID date	character vector
VT_INT VT_UINT VT_I2 VT_UI2 VT_I4 VT_UI4 VT_R4 VT_R8 VT_CY	signed machine int unsigned machine int 2-byte signed int unsigned short 4-byte signed int unsigned long 4-byte real 8-byte real currency	double
VT_I8	signed int64	int64
VT_UI8	unsigned int64	uint64
VT_BOOL		logical
VT_I1 VT_UI1	signed char unsigned char	char
VT_VARIANT VT_USERDEFINED VT_ARRAY	VARIANT * user-defined type SAFEARRAY*	variant
VT_VARIANT VT_BYREF	VARIANT * void* for local use	variant(pointer)
VT_SAFEARRAY	use VT_ARRAY in VARIANT	SAFEARRAY

COM Variant Type	Description	MATLAB Representation
VT_SAFEARRAY VT_BYREF		SAFEARRAY(pointer)

Unsupported Types

MATLAB does not support the following COM interface types.

- Structure
- Sparse array
- Multidimensional SAFEARRAYs (greater than two dimensions)
- Write-only properties

Pass MATLAB SAFEARRAY to COM Object

The SAFEARRAY data type is a standard way to pass arrays between COM objects. This section explains how MATLAB passes SAFEARRAY data to a COM object.

- “Default Behavior in MATLAB” on page 17-7
- “Examples” on page 17-7
- “Pass Single-Dimension SAFEARRAY” on page 17-9
- “Pass SAFEARRAY by Reference” on page 17-9

Default Behavior in MATLAB

MATLAB represents an m -by- n matrix as a two-dimensional SAFEARRAY, where the first dimension has m elements and the second dimension has n elements. MATLAB passes the SAFEARRAY by value.

Examples

The following examples use a COM object that expects a SAFEARRAY input parameter.

When MATLAB passes a 1-by-3 array:

```
B = [2 3 4]
```

```
B =
     2     3     4
```

the object reads:

```
No. of dimensions: 2
Dim: 1,          No. of elements: 1
Dim: 2,          No. of elements: 3
                Elements:
                2.0
                3.0
                4.0
```

When MATLAB passes a 3-by-1 array:

```
C = [1;2;3]
C =
     1
     2
     3
```

the object reads:

```
No. of dimensions: 2
Dim: 1,          No. of elements: 3
Dim: 2,          No. of elements: 1
Elements:
        1.0
        2.0
        3.0
```

When MATLAB passes a 2-by-4 array:

```
D = [2 3 4 5;5 6 7 8]
D =
     2     3     4     5
     5     6     7     8
```

the object reads:

```
No. of dimensions: 2
Dim: 1,          No. of elements: 2
Dim: 2,          No. of elements: 4
Elements:
        2.0
        3.0
        4.0
        5.0
        5.0
```

6.0
7.0
8.0

Pass Single-Dimension SAFEARRAY

For information, see [How can I pass arguments to an ActiveX server from MATLAB 7.0 \(R14\) as one-dimensional arrays?](#)

Pass SAFEARRAY by Reference

For information, see [How can I pass arguments by reference to an ActiveX server from MATLAB 7.0 \(R14\)?](#)

Read SAFEARRAY from COM Objects in MATLAB Applications

This section explains how MATLAB reads SAFEARRAY data from a COM object.

MATLAB reads a one-dimensional SAFEARRAY with n elements from a COM object as a 1-by- n matrix.

MATLAB reads a two-dimensional SAFEARRAY with n elements as a 2-by- n matrix.

MATLAB reads a three-dimensional SAFEARRAY with two elements as a 2-by-2-by-2 cell array.

Display MATLAB Syntax for COM Objects

To determine which MATLAB types to use when passing arguments to COM objects, use the `invoke` or `methodsview` functions. These functions list all the methods found in an object, along with a specification of the types required for each argument.

Consider a server called `MyApp`, which has a single method `TestMeth1` with the following syntax:

```
HRESULT TestMeth1 ([out, retval] double* dret);
```

This method has no input argument, and it returns a variable of type `double`. The following **pseudo-code** displays the MATLAB syntax for calling the method.

```
h = actxserver('MyApp');  
invoke(h)
```

MATLAB displays:

```
ans =  
    TestMeth1 = double TestMeth1 (handle)
```

The signature of TestMeth1 is:

```
double TestMeth1(handle)
```

MATLAB requires you to use an object handle as an input argument for every method, in addition to any input arguments required by the method itself.

Use one of the following **pseudo-code** commands to create the variable `var`, which is of type `double`.

```
var = h.TestMeth1;
```

or:

```
var = TestMeth1(h);
```

Although the following syntax is correct, its use is discouraged:

```
var = invoke(h, 'TestMeth1');
```

Now consider the server called `MyApp1` with the following methods:

```
HRESULT TestMeth1 ([out, retval] double* dret);  
HRESULT TestMeth2 ([in] double* d, [out, retval] double* dret);  
HRESULT TestMeth3 ([out] BSTR* sout,  
                  [in, out] double* dinout,  
                  [in, out] BSTR* sinout,  
                  [in] short sh,  
                  [out] long* ln,  
                  [in, out] float* bl,  
                  [out, retval] double* dret);
```

Using the `invoke` function, MATLAB displays the list of methods:

```
ans =  
    TestMeth1 = double TestMeth1 (handle)  
    TestMeth2 = double TestMeth2 (handle, double)  
    TestMeth3 = [double, string, double, string, int32, single] ...  
                TestMeth3(handle, double, string, int16, single)
```

`TestMeth2` requires an input argument `d` of type `double`, and returns a variable `dret` of type `double`. Some **pseudo-code** examples of calling `TestMeth2` are:


```
var = h.TestMeth2(5);
```

or:

```
var = TestMeth2(h, 5);
```

`TestMeth3` requires multiple input arguments, as indicated within the parentheses on the right side of the equal sign, and returns multiple output arguments, as indicated within the brackets on the left side of the equal sign.

```
[double, string, double, string, int32, single] %output arguments
TestMeth3(handle, double, string, int16, single) %input arguments
```

The first input argument is the required `handle`, followed by four input arguments.

```
TestMeth3(handle, in1, in2, in3, in4)
```

The first output argument is the return value `retval`, followed by five output arguments.

```
[retval, out1, out2, out3, out4, out5]
```

This is how the arguments map into a MATLAB command:

```
[dret, sout, dinout, sinout, ln, b1] = TestMeth3(handle, ...
                                             dinout, sinout, sh, b1)
```

where `dret` is `double`, `sout` is `string`, `dinout` is `double` and is both an input and an output argument, `sinout` is `string` (input and output argument), `ln` is `int32`, `b1` is `single` (input and output argument), `handle` is the handle to the object, and `sh` is `int16`.

COM Object Properties

MATLAB Functions for Object Properties

You can get the value of a property and, sometimes, change the value. You also can add custom properties.

Property names are not case-sensitive. You can abbreviate them as long as the name is unambiguous.

Function	Description
<code>get</code>	List one or more properties and their values.
<code>set</code>	Set the value of one or more properties.
<code>isprop</code>	Determine if an item is a property of a COM object.
<code>addproperty</code>	Add a custom property to a COM object.
<code>deleteproperty</code>	Remove a custom property from a COM object.
<code>inspect</code>	Open the Property Inspector to display and modify property values.
<code>propedit</code>	Display the built-in property page of the control, if any.

Work with Multiple Objects

You can use the `get` and `set` functions on more than one object at a time by creating a vector of object handles and using these commands on the vector. To get or set values for multiple objects, use the functional form of the `get` and `set` functions. Use dot notation, for example `h.propname`, on scalar objects only.

Enumerated Values for Properties

Enumeration makes examining and changing properties easier because each possible value for the property is assigned text to represent it. For example, one of the values for the `DefaultSaveFormat` property in a Microsoft Excel spreadsheet is `xlUnicodeText`. This text is easier to remember than a numeric value like 57.

Property Inspector

The Property Inspector enables you to access the properties of COM objects. To open the Property Inspector, use the `inspect` function from the MATLAB command line or double-click the object in the MATLAB Workspace browser.

For example, create an Excel object. Then set the `DefaultFilePath` property to an existing folder, `C:\ExcelWork`.

```
h = actxserver('Excel.Application');  
h.DefaultFilePath = 'C:\ExcelWork';
```

Display the properties of the object.

```
inspect(h)
```

Scroll down until you see the `DefaultFilePath` property that you just changed, `C:\ExcelWork`.

Using the Property Inspector, change `DefaultFilePath` once more, this time to another existing folder, `MyWorkDirectory`. To do this, select the value at the right and type the new value.

Now go back to the MATLAB Command Window and confirm that the `DefaultFilePath` property has changed as expected.

```
h.DefaultFilePath
```

```
ans =
```

```
C:\MyWorkDirectory
```

Note If you modify properties at the MATLAB command line, refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector window by reinvoking the `inspect` function on the object.

Enumerated Values

A list button next to a property value indicates that the property accepts enumerated values. To see the values, click anywhere in the field on the right. For example, the `Cursor` property has four enumerated values. The current value `xlDefault` is displayed in the field next to the property name.

To change the value, use the list button to display the options for that property, and then click the desired value.

Custom Properties

You can add your own properties to an instance of a control using the `addproperty` function.

To remove custom properties from a control, use the `deleteproperty` function.

Properties That Take Arguments

Some COM objects have properties that accept input arguments. Internally, MATLAB handles these properties as methods, which means you use the `methods` or `invoke` functions (not the `get` function) to view the property.

See Also

Related Examples

- “Change Cursor in Spreadsheet” on page 17-36
- “Change Row Height in Range of Spreadsheet Cells” on page 17-31

More About

- “Properties” on page 17-27

COM Methods

In this section...
“Method Information” on page 17-15
“Call Object Methods” on page 17-15
“Specify Enumerated Parameters” on page 17-16
“Skip Optional Input Arguments” on page 17-16
“Return Multiple Output Arguments” on page 17-17

Method Information

You execute, or invoke, COM functions or methods belonging to COM objects. Method names are case-sensitive. You cannot abbreviate them.

To see what methods a COM object supports, use one of the following functions. Each function presents specific information, as described in the table. For information about using a method, refer to your vendor documentation.

Function	Output
methodsvIEW	Graphical display of function names and signatures
methods with -full qualifier	Cell array of function names and signatures, sorted alphabetically
methods	Cell array of function names only, sorted alphabetically, with uppercase names listed first
invoke	Cell array of function names and signatures

Call Object Methods

MATLAB supports the following syntaxes to call methods on an object.

- By method name:


```
outputvalue = methodname(object, 'arg1', 'arg2', ...);
```
- By dot notation:

```
outputvalue = object.methodname('arg1','arg2',...);
```

- Using explicit syntax:

```
outputvalue = invoke(object,'methodname','arg1','arg2',...);
```

The `methodsview` and `methods -full` commands show what data types to use for input and output arguments.

You cannot use dot syntax and must explicitly call the `get`, `set`, and `invoke` functions under the following conditions:

- To access a property or method that is not a public member of the object class.
- To access a property or method that is not in the type library for the server.
- To access properties that take arguments. MATLAB treats these properties like methods.
- To access properties on a vector of objects, use the `get` and `set` functions.

You cannot invoke a method on multiple COM objects, even if you call the `invoke` function explicitly.

Specify Enumerated Parameters

Enumeration is a way of assigning a descriptive name to a symbolic value. MATLAB supports enumeration for parameters passed to methods under the condition that the type library in use reports the parameter as `ENUM`, and only as `ENUM`.

Note MATLAB does not support enumeration for any parameter that the type library reports as both `ENUM` and `Optional`.

Skip Optional Input Arguments

When calling a method that takes optional input arguments, you can skip an optional argument by specifying an empty array (`[]`) in its place. For example, the syntax for calling a method with second argument `arg2` not specified is:

```
methodname(handle, arg1, [], arg3);
```

Return Multiple Output Arguments

If a server function supports multiple outputs, you can return any or all those outputs to a MATLAB client.

The following syntax shows a server function `functionname` called by the MATLAB client. `retval` is the first output argument, or return value. The other output arguments are `out1`, `out2`,

```
[retval out1 out2 ...] = functionname(handle,in1,in2,...);
```

MATLAB uses the pass-by-reference capabilities in COM to implement this feature. Pass-by-reference is a COM feature; MATLAB does not support pass-by-reference.

See Also

Related Examples

- “Change Row Height in Range of Spreadsheet Cells” on page 17-31
- “Insert Spreadsheet After First Sheet” on page 17-38

More About

- “Handle COM Data in MATLAB” on page 17-4

COM Events

An *event* is typically a user-initiated action that takes place in a server application, which often requires a reaction from the client. For example, if you click the mouse at a particular location in a server interface window, the client application can respond. When an event is *fired*, the server communicates this occurrence to the client. If the client is *listening* for this particular type of event, it responds by executing a routine called an event handler.

The MATLAB COM client can subscribe to and handle the events fired by a COM server. Select the events you want the client to listen to. Register each event with an event handler to be used in responding to the event. When a registered event takes place, the server notifies the client, which responds by executing the appropriate event handler routine. You can write event handlers as MATLAB functions.

To identify events the server can respond to, use the `events` function.

To register events you want to respond to, use the `registerevent` function. The MATLAB client responds only to events you have registered. If you register the same event name to the same callback handler multiple times, MATLAB executes the event only once.

To identify registered events, use the `eventlisteners` function.

To respond to events as they occur, create event handlers that have been registered for that event. You can implement these routines as MATLAB functions.

To unregister events you no longer want to listen to, use the `unregisterevent` or `unregisterallevents` function.

Note MATLAB does not support asynchronous events.

Note MATLAB does not support interface events from a Custom server.

See Also

`eventlisteners` | `events` | `registerevent` | `unregisterallevents` | `unregisterevent`

Related Examples

- “Read Spreadsheet Data Using Excel as Automation Server” on page 16-6

More About

- “COM Event Handlers” on page 17-20

COM Event Handlers

Use `registerevent` to register server events. Use `events` to list all the events a COM object recognizes.

Arguments Passed to Event Handlers

When a registered event is triggered, MATLAB passes information from the event to its handler function, as shown in the following table.

Arguments Passed by MATLAB Functions

Arg. No.	Contents	Format
1	Object name	MATLAB COM class
2	Event ID	double
3	Start of Event Argument List	As passed by the control
end - 2	End of Event Argument List (Argument N)	As passed by the control
end - 1	Event Structure	structure
end	Event Name	char array

When writing an event handler function, use the Event Name argument to identify the source of the event. Get the arguments passed by the control from the Event Argument List (arguments 3 through end-2). All event handlers must accept a variable number of arguments:

```
function event(varargin)
if (strcmp(varargin{end}, 'MouseDown')) % Check the event name
    x_pos = varargin{5}; % Read 5th Event Argument
    y_pos = varargin{6}; % Read 6th Event Argument
end
```

Note The values passed vary with the particular event and control being used.

Event Structure

The Event Structure argument passed by MATLAB contains the fields shown in the following table.

Fields of the Event Structure

Field Name	Description	Format
Type	Event Name	char array
Source	Control Name	MATLAB COM class
EventID	Event Identifier	double
Event Arg Name 1	Event Arg Value 1	As passed by the control
Event Arg Name 2	Event Arg Value 2	As passed by the control
etc.	Event Arg N	As passed by the control

See Also

events | registerevent

More About

- “COM Events” on page 17-18

COM Object Interfaces

IUnknown and IDispatch Interfaces

When you invoke the `actxserver` function, MATLAB creates the server and returns a handle to the server interface as a means of accessing its properties and methods. The software uses the following process to determine which handle to return:

- 1 First get a handle to the IUnknown interface from the component. All COM components are required to implement this interface.
- 2 Attempt to get the IDispatch interface. If IDispatch is implemented, return a handle to this interface. If IDispatch is not implemented, return the handle to IUnknown.

Additional Interfaces

Components often provide additional interfaces, based on IDispatch, that are implemented as properties. Like any other property, you obtain these interfaces using the MATLAB `get` function.

For example, a Microsoft Excel component contains numerous interfaces. To list these interfaces, along with Excel properties, type:

```
h = actxserver('Excel.Application');  
get(h)
```

MATLAB displays information like:

```
Application: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library_Application]
  Creator: 'xlCreatorCode'
  Parent: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library_Application]
  ActiveCell: []
  ActiveChart: [1x50 char]
  .
  .
```

To see if `Workbooks` is an interface, type:

```
w = h.Workbooks
```

MATLAB displays:

`W =`
`Interface.Microsoft_Excel_9.0_Object_Library.Workbooks`

The information displayed depends on the version of the Excel software you have on your system.

For examples using Excel in MATLAB, see:

- “Write Data to Excel Spreadsheet” on page 17-34
- “Read Spreadsheet Data Using Excel as Automation Server” on page 16-6
- `actxserver`

See Also

More About

- “Supported Client/Server Configurations” on page 16-13

Save and Delete COM Objects

Functions to Save and Restore COM Objects

Use these MATLAB functions to save and restore the state of a COM control object.

Function	Description
load	Load and initialize a COM control object from a file
save	Write and serialize a COM control object to a file

Save, or serialize, the current state of a COM control to a file using the `save` function. *Serialization* is the process of saving an object onto a storage medium (such as a file or a memory buffer) or transmitting it across a network connection link in binary form.

Note MATLAB supports the COM `save` and `load` functions for controls only.

Release COM Interfaces and Objects

Use these MATLAB functions to release or delete a COM object or interface.

Function	Description
delete	Delete a COM object or interface
release	Release a COM interface

When you no longer need an interface, use the `release` function to release the interface and reclaim the memory used by it. When you no longer need a server, use the `delete` function to delete it. Alternatively, you can use the `delete` function to both release all interfaces for the object and delete the server or control.

Note In versions of MATLAB earlier than 6.5, failure to explicitly release interface handles or delete the server often results in a memory leak. This is true even if the variable representing the interface or COM object has been reassigned. In MATLAB version 6.5 and later, the server, along with all interfaces to it, is destroyed on reassignment of the variable or when the variable representing a COM object or interface goes out of scope.

When you delete or close a figure window containing a control, MATLAB automatically releases all interfaces for the control. MATLAB also automatically releases all handles for an Automation server when you exit the program.

See Also

MATLAB Application as DCOM Client

Distributed Component Object Model (DCOM) is a protocol that allows clients to use remote COM objects over a network. Also, MATLAB can be used as a DCOM client with remote Automation servers if the operating system on which MATLAB is running is DCOM enabled.

Note If you use MATLAB as a remote DCOM server, all MATLAB windows appears on the remote machine.

Explore COM Objects

A COM object has properties, methods, events, and interfaces. Your vendor documentation describes these features, but you can also learn about your object using MATLAB commands.

Properties

A property is information that is associated with a COM object. To see a list of properties of an object, use the `get` function. Alternatively, use the MATLAB Property Inspector, a user interface to display and modify properties. For example, to list all properties of a Microsoft Excel object type the following command. MATLAB displays the properties for your Excel version.

```
myApp = actxserver('Excel.Application');  
get(myApp)
```

To display a single property, type the following. MATLAB displays the value for your application.

```
myApp.OrganizationName  
  
ans =  
  
MathWorks, Inc.
```

To open the Property Inspector, choose one of the following. MATLAB opens the Inspector window.

- Call the `inspect` function from the MATLAB command line:

```
inspect(myApp)
```
- Double-click the `myApp` object in the MATLAB Workspace browser.

Scroll down until you see the `OrganizationName` property, the same value returned by the `get` function.

Methods

A method is a procedure you call to perform a specific action on the COM object. For example, to list all methods supported by the Excel object, type the following. MATLAB opens a window showing the method signatures for `COM.Excel_Application` objects.

```
myApp = actxserver('Excel.Application');  
methodsviw(myApp)
```

Events

An *event* is typically a user-initiated action that takes place in a server application, which often requires a reaction from the client. For example, clicking the mouse at a particular location in a server interface window might require the client to respond. When an event is *fired*, the server communicates this occurrence to the client. If the client is *listening* for this particular type of event, it responds by executing a routine called an event handler.

Use the `events` function to list all events known to the server and use the `eventlisteners` function to list registered events.

For example, to list the events for the Microsoft Internet Explorer web browser, type the following. MATLAB displays the events for your Internet Explorer version.

```
myNet = actxserver('internetexplorer.application');  
events(myNet)
```

To see which events have event handlers, type:

```
eventlisteners(myNet)  
  
ans =  
    {}
```

An empty result means that no events are registered.

Interfaces

An interface is a set of related functions used to access the data of a COM object. When you create a COM object using the `actxserver` function, MATLAB returns a handle to an interface. Use the `get` and `interfaces` functions to see other interfaces implemented by your object.

For example, to see interfaces of an Excel object, type:

```
e = actxserver('Excel.Application');  
get(e)
```

MATLAB displays the properties, including interfaces, for your Excel version. For example, `Workbooks` is an interface.

e.Workbooks

```
ans =
```

```
Interface.000208DB_0000_0000_C000_000000000046
```

To explore the Workbooks interface, create a workbooks object and use the relevant MATLAB commands.

```
w = e.Workbooks;
```

Identify Objects and Interfaces

Function	Description
<code>class</code>	Return the class of an object.
<code>isa</code>	Determine if an object is of a given MATLAB class.
<code>iscom</code>	Determine if the input is a COM object.
<code>isevent</code>	Determine if an item is an event of a COM object.
<code>ismethod</code>	Determine if an item is a method of a COM object.
<code>isprop</code>	Determine if an item is a property of a COM object.
<code>isinterface</code>	Determine if the input is a COM interface.

See Also

Property Inspector | [eventlisteners](#) | [events](#) | [get](#) | [methodsview](#)

More About

- “COM Object Properties” on page 17-12
- “Property Inspector” on page 17-13
- “COM Methods” on page 17-15
- “COM Events” on page 17-18
- “COM Event Handlers” on page 17-20

External Websites

- Microsoft Excel 2013 developer reference

Change Row Height in Range of Spreadsheet Cells

This example shows how to change the height of a row, defined by a Range object, in a spreadsheet.

The Excel® Range object is a property that takes input arguments. MATLAB® treats such a property as a method. Use the methods function to get information about creating a Range object.

Create a Worksheet object ws .

```
e = actxserver('Excel.Application');
wb = Add(e.Workbooks);
e.Visible = 1;
ws = e.ActiveSheet;
```

Display the default height of all the rows in the worksheet.

```
ws.StandardHeight
```

```
ans =
    14.4000
```

Display the function syntax for creating a Range object. Search the displayed list for the Range entry: handle Range(handle,Variant,Variant(Optional))

```
methods(ws, '-full')
```

```
Methods for class Interface.000208D8_0000_0000_C000_000000000046:
```

```
Activate(handle)
Calculate(handle)
handle ChartObjects(handle, Variant(Optional))
CheckSpelling(handle, Variant(Optional))
CircleInvalid(handle)
ClearArrows(handle)
ClearCircles(handle)
Copy(handle, Variant(Optional))
Delete(handle)
Variant Evaluate(handle, Variant)
```

```
ExportAsFixedFormat(handle, XlFixedFormatType, Variant(Optional))
Move(handle, Variant(Optional))
handle OLEObjects(handle, Variant(Optional))
Paste(handle, Variant(Optional))
PasteSpecial(handle, Variant(Optional))
handle PivotTableWizard(handle, Variant(Optional))
handle PivotTables(handle, Variant(Optional))
PrintOut(handle, Variant(Optional))
PrintPreview(handle, Variant(Optional))
Protect(handle, Variant(Optional))
handle Range(handle, Variant, Variant(Optional))
ResetAllPageBreaks(handle)
SaveAs(handle, ustring, Variant(Optional))
handle Scenarios(handle, Variant(Optional))
Select(handle, Variant(Optional))
SetBackgroundPicture(handle, ustring)
ShowAllData(handle)
ShowDataForm(handle)
Unprotect(handle, Variant(Optional))
handle XmlDataQuery(handle, ustring, Variant(Optional))
handle XmlMapQuery(handle, ustring, Variant(Optional))
addproperty(handle, string)
delete(handle, MATLAB array)
deleteproperty(handle, string)
MATLAB array events(handle, MATLAB array)
MATLAB array get(handle)
MATLAB array get(handle, MATLAB array, MATLAB array)
MATLAB array get(handle vector, MATLAB array, MATLAB array)
MATLAB array invoke(handle)
MATLAB array invoke(handle, string, MATLAB array)
MATLAB array loadobj(handle)
release(handle, MATLAB array)
MATLAB array saveobj(handle)
MATLAB array set(handle vector, MATLAB array, MATLAB array)
MATLAB array set(handle, MATLAB array, MATLAB array)
MATLAB array set(handle)
```

Create a Range object consisting of the first row.

```
wsRange = Range(ws, 'A1');
```

Increase the row height.

```
wsRange.RowHeight = 25;
```

Open the worksheet, click in row 1, and notice the height.

Close the workbook without saving.

```
wb.Saved = 1;  
Close(e.Workbook)
```

Close the application.

```
Quit(e)  
delete(e)
```

See Also

methods

More About

- “Properties That Take Arguments” on page 17-14

External Websites

- Worksheet.Range Property (Excel) Office 2013

Write Data to Excel Spreadsheet

This example shows how to write a MATLAB matrix to an Excel spreadsheet. For alternatives to exporting MATLAB data to a Microsoft Excel spreadsheet, see the functions and examples in “Spreadsheets”.

Create an Excel object.

```
e = actxserver('Excel.Application');
```

Add a workbook.

```
eWorkbook = e.Workbooks.Add;  
e.Visible = 1;
```

Make the first sheet active.

```
eSheets = e.ActiveWorkbook.Sheets;  
eSheet1 = eSheets.get('Item',1);  
eSheet1.Activate
```

Put MATLAB data into the worksheet.

```
A = [1 2; 3 4];  
eActivesheetRange = get(e.Activesheet, 'Range', 'A1:B2');  
eActivesheetRange.Value = A;
```

Read the data back into MATLAB, where array B is a cell array.

```
eRange = get(e.Activesheet, 'Range', 'A1:B2');  
B = eRange.Value;
```

Convert the data to a double matrix. Use the following command if the cell array contains only scalar values.

```
B = reshape([B{:}], size(B));
```

Save the workbook in a file.

```
SaveAs(eWorkbook, 'myfile.xls')
```

If the Excel program displays a dialog box about saving the file, select the appropriate response to continue.

If you saved the file, then close the workbook.


```
eWorkbook.Saved = 1;  
Close(eWorkbook)
```

Quit the Excel program and delete the server object.

```
Quit(e)  
delete(e)
```

Note Make sure that you close workbook objects you create to prevent potential memory leaks.

See Also

xlswrite

More About

- “Read Collection or Sequence of Spreadsheet Files”

External Websites

- Microsoft Excel 2013 developer reference

Change Cursor in Spreadsheet

This example shows how to change the cursor icon in an Excel® spreadsheet.

Create a COM server running a Microsoft® Excel application.

```
h = actxserver('Excel.Application');  
h.Visible = 1;
```

Open the Excel program and notice the cursor.

View the options for the `Cursor` property. These values are enumerated values, which means they are the only values allowed for the property.

```
set(h, 'Cursor')
```

```
ans =
```

```
4×1 cell array  
  
'xlIBeam'  
'xlDefault'  
'xlNorthwestArrow'  
'xlWait'
```

Change the cursor to `xlIBeam`.

```
h.Cursor = 'xlIBeam';
```

Notice the cursor in the program.

Reset the cursor.

```
h.Cursor = 'xlDefault';
```

Close workbook objects you created to prevent potential memory leaks.

Quit(h)
delete(h)

See Also

More About

- “Enumerated Values for Properties” on page 17-12

Insert Spreadsheet After First Sheet

This example shows how to skip an optional input argument in the Excel Add method, used to insert a sheet into a workbook.

The Add method has the following optional input arguments:

- **Before** — The sheet before which to add the new sheet
- **After** — The sheet after which to add the new sheet
- **Count** — The total number of sheets to add
- **Type** — The type of sheet to add

```
e = actxserver('Excel.Application');  
Add(e.Workbooks);  
e.Visible = 1;
```

Create a collection of the default sheets in the workbook.

```
eSheets = e.ActiveWorkbook.Sheets;
```

Insert a sheet after the first item in the collection, `eSheet1`.

```
eSheet1 = Item(eSheets,1);  
eNewSheet = Add(eSheets, [], eSheet1);
```

To call Add with the **After** argument, omit the first argument, **Before**, by using an empty array `[]` in its place.

Open the workbook and notice Sheet4.

Close the spreadsheet.

Close the application.

Quit(e)
delete(e)

See Also

More About

- “Skip Optional Input Arguments” on page 17-16

External Websites

- Office 2013 Sheets.Add Method (Excel)

Connect to Existing Excel Application

This example shows how to read data from an open file, `weekly_log.xlsx`, in MATLAB.

MATLAB can access a file that is open by another application by creating a COM server from the MATLAB client, and then opening the file through this server.

Navigate to a folder containing an Excel file, for example, `weekly_log.xlsx`. Open the file in the Excel program.

Open the same file in MATLAB.

```
excelapp = actxserver('Excel.Application');  
wkbk = excelapp.Workbooks;  
wdata = Open(wkbk, 'c:\work\weekly_log.xlsx');
```

Read data in the range D1 and F6 from sheet 2.

```
sheets = wdata.Sheets;  
sheet12 = Item(sheets,2);  
range = get(sheet12, 'Range', 'D1', 'F6');  
range.value
```

```
ans =
```

'Temp.'	'Heat Index'	'Wind Chill'
[78.4200]	[32]	[37]
[69.7300]	[27]	[30]
[77.6500]	[17]	[16]
[74.2500]	[-5]	[0]
[68.1900]	[22]	[35]

```
Close(wkbk)  
Quit(excelapp)
```

See Also

`actxserver`

Display Message for Workbook OnClose Event

This example shows how to handle a COM interface event, how to set up an event in a Microsoft Excel workbook object, and how to handle its BeforeClose event.

Create the following event handler file, OnBeforeCloseWorkbook.m, in your current folder.

```
function OnBeforeCloseWorkbook(varargin)
disp('BeforeClose event occurred')
```

Create the Excel object and make it visible.

```
xl = actxserver('Excel.Application');
xl.Visible = 1;
```

Add a workbook.

```
hWbks = xl.Workbooks;
hWorkbook = hWbks.Add;
```

Register the OnBeforeCloseWorkbook function for the OnClose event.

```
registerevent(hWorkbook,{'BeforeClose' @OnBeforeCloseWorkbook})
```

Close the workbook, which triggers the Close event and calls the OnClose handler.

```
Close(hWorkbook)
```

```
BeforeClose event occurred
```

```
Quit(xl)
```

See Also

registerevent

More About

- “COM Events” on page 17-18

COM Collections

COM *collections* are a way to support groups of related COM objects that can be iterated over. A collection is itself an interface with a read-only `Count` property and an `Item` method to retrieve a single item from the collection.

The `Item` method is indexed, which means that it requires an argument that specifies which item in the collection is being requested. The data type of the index is a data type specified by the server that supports the collection. Although integer indices are common, the index could also be a text value. Often, the return value from the `Item` method is itself an interface. Like all interfaces, release this interface when you are finished with it.

This example iterates through the members of a collection. Each member of the collection is itself an interface (called `Plot` and represented by a MATLAB COM object called `hPlot`). In particular, this example iterates through a collection of `Plot` interfaces, invokes the `Redraw` method for each interface, and then releases each interface:

```
hCollection = hControl.Plots;
for i = 1:hCollection.Count
    hPlot = invoke(hCollection,'Item', i);
    Redraw(hPlot)
    release(hPlot);
end;
release(hCollection);
```


MATLAB COM Support Limitations

Microsoft does not support loading 32-bit DLLs or in-process COM servers into a 64-bit application, or conversely. So you cannot use 32-bit DLL COM objects in 64-bit MATLAB.

Limitations of MATLAB COM support are:

- MATLAB does not support custom COM interfaces with 64-bit MATLAB.
- MATLAB only supports indexed collections.
- COM controls are not printed with figure windows.
- “Unsupported Types” on page 17-7
- MATLAB does not support asynchronous events.

Interpreting Argument Callouts in COM Error Messages

When a MATLAB client sends a command with an invalid argument to a COM server application, the server sends back an error message in the following format.

```
??? Error: Type mismatch, argument n.
```

If you do not use the dot syntax format, be careful interpreting the argument number in this message.

For example, using dot syntax, if you type:

```
handle.PutFullMatrix('a','base',7,[5 8]);
```

MATLAB displays:

```
??? Error: Type mismatch, argument 3.
```

In this case, the argument, 7, is invalid because `PutFullMatrix` expects the third argument to be an array data type, not a scalar. In this example, the error message identifies 7 as argument 3.

However, if you use the syntax:

```
PutFullMatrix(handle,'a','base',7,[5 8]);
```

MATLAB displays:

```
??? Error: Type mismatch, argument 3.
```

In this call to the `PutFullMatrix` function, 7 is argument four. However, the COM server does not receive the first argument. The `handle` argument merely identifies the server. It does not get passed to the server. The server reads 'a' as the first argument, and the invalid argument, 7, as the third.

See Also

More About

- “COM Methods” on page 17-15

MATLAB COM Automation Server Support

Register MATLAB as Automation Server

When to Register MATLAB

When you install a new version of MATLAB, MATLAB automatically registers this version as an Automation server. You might have multiple versions of MATLAB installed on your system, but only one version is registered. This version remains registered until you install a new MATLAB or you register a different version.

If the registered version of MATLAB is not the preferred version, then choose one of these techniques.

- “Register from System Prompt” on page 18-2
- “Register from MATLAB Command Prompt” on page 18-3

You must have administrator privileges to change the Windows registry. Based on your User Account Control (UAC) settings, you might need to right-click the Windows Command Processor or the MATLAB icon and select **Run as administrator**. If that option is not available, contact your system administrator.

If MATLAB is unable to update the registry, it still displays a new minimized command window. Open this window and exit MATLAB.

Register from System Prompt

To register MATLAB as an Automation server from the Windows system prompt, first open a Windows Command Processor using the **Run as administrator** option, if necessary.

Move to the folder containing the executable file for the MATLAB version you want.

```
cd matlabroot\bin\arch
```

If you do not use this folder, the `matlab` command starts the first instance of MATLAB on the system path.

```
matlab -regserver
```

MATLAB displays a minimized command window. Open this window and exit MATLAB.

Register from MATLAB Command Prompt

To register MATLAB as an Automation server from within MATLAB, first start MATLAB with the **Run as administrator** option, if necessary. When you run this command, you register this version of MATLAB.

```
regmatlabserver
```

MATLAB displays a minimized command window. Restore this window and exit MATLAB.

Unregister MATLAB as Automation Server

At the operating system prompt, navigate to the MATLAB folder as described in “Register from System Prompt” on page 18-2 and type:

```
matlab -unregserver
```

See Also

matlab (Windows) | regmatlabserver

More About

- “Create MATLAB Server” on page 18-7

MATLAB COM Automation Server Interface

COM Server Types

- Automation — A server that supports the OLE Automation standard. Automation servers are based on the `IDispatch` interface. Clients of all types, including scripting clients, access Automation servers.
- Custom — A server that implements an interface directly derived from `IUnknown`. MATLAB does not support custom interfaces.
- Dual — A server that implements a combination of Automation and Custom interfaces.

Programmatic Identifiers

To create an instance of a COM object, you use its programmatic identifier, or ProgID. The ProgID is a unique string defined by the component vendor to identify the COM object. You obtain a ProgID from your vendor documentation.

The MATLAB ProgIDs for shared servers are:

- `Matlab.Application` — Starts a command window Automation server with the version of MATLAB that was most recently used as an Automation server (might not be the latest installed version of MATLAB).
- `Matlab.Autoserver` — Starts a command window Automation server using the most recent version of MATLAB.
- `Matlab.Desktop.Application` — Starts the full desktop MATLAB as an Automation server using the most recent version of MATLAB.

The ProgIDs for dedicated servers are:

- `Matlab.Application.Single`
- `Matlab.Autoserver.Single`

These version-independent MATLAB ProgIDs specify the currently installed and registered version of MATLAB.

In order to create an instance of a specific installed and registered MATLAB version, you can use a version-dependent ProgID. For example, using the ProgID `Matlab.Application.7.14` creates an instance of MATLAB version 7.14 (R2012a).

Shared and Dedicated Servers

You can start the MATLAB Automation server in one of two modes - shared or dedicated. A dedicated server is dedicated to a single client; a shared server is shared by multiple clients. The mode is determined by the programmatic identifier (ProgID) used by the client to start MATLAB. If you use `matlab.application` as your ProgID, then MATLAB creates a shared server.

Starting a Shared Server

The ProgID, `matlab.application`, specifies the default mode, which is shared. You can also use the version-specific ProgID, `matlab.application.N.M`, where `N` is the major version and `M` is the minor version of your MATLAB. For example, use `N = 7` and `M = 4` for MATLAB version 7.4.

Once MATLAB is started as a shared server, all clients that request a connection to MATLAB using the shared server ProgID connect to the already running instance of MATLAB. In other words, there is never more than one instance of a shared server running, since it is shared by all clients that use the shared server ProgID.

Starting a Dedicated Server

To specify a dedicated server, use the ProgID, `matlab.application.single`, (or the version-specific ProgID, `matlab.application.single.N.M`).

Each client that requests a connection to MATLAB using a dedicated ProgID creates a separate instance of MATLAB; it also requests the server not be shared with any other client. Therefore, there can be several instances of a dedicated server running simultaneously, since the dedicated server is not shared by multiple clients.

In-Process and Out-of-Process Servers

MATLAB supports the following server configurations.

In-Process Server

An in-process server is a component implemented as a dynamic link library (DLL) that runs in the same process as the client application, sharing address space. Communication between client and server is relatively fast and simple.

Local Out-of-Process Server

A local out-of-process server is a component implemented as an executable (EXE) file that runs in a separate process from the client application. The client and server processes are on the same computer system. This configuration is slower due to the overhead required when transferring data across process boundaries.

Remote Out-of-Process Server

Distributed Component Object Model (DCOM) is a protocol that allows COM connections to be established over a network. If you are using a version of the Windows operating system that supports DCOM and a controller that supports DCOM, then you can use the controller to start a MATLAB server on a remote machine. DCOM must be configured properly, and MATLAB must be installed on each machine that is used as a client or server. Even though the client machine might not run MATLAB in such a configuration, the client machine must have a MATLAB installation because certain MATLAB components are required to establish the remote connection. Consult your DCOM documentation for how to configure DCOM for your environment.

Network communications, in addition to the overhead required for data transfer, can make this configuration slower than the local out-of-process configuration.

For more information, see the following articles.

- How can I make a DCOM Server instance of MATLAB visible?
- How can I utilize MATLAB on a remote machine as a Distributed COM (DCOM) server?

Create MATLAB Server

Getting Started

To create a server, you need a programmatic identifier (ProgID) to identify the server. The ProgID for MATLAB is `matlab.application`. For a complete list, see “Programmatic Identifiers” on page 18-4.

How you create an Automation server depends on the controller you are using. Consult your controller's documentation for this information. Possible options include:

- Add this statement to a Visual Basic .NET application.

```
MatLab = CreateObject("MatLab.Application")
```
- “Manually Create Automation Server” on page 18-13.
- Use the Visual Basic Object Browser to access methods.

Access Methods from the Visual Basic Object Browser

You can use the Object Browser of your Visual Basic client application to see what methods are available from a MATLAB Automation server. To do this you need to reference the MATLAB *type library* in your Visual Basic project.

To set up your Visual Basic project:

- 1 Select the **Project** menu.
- 2 Select **Reference** from the subsequent menu.
- 3 Check the box next to the **MATLAB Application Type Library**.
- 4 Click **OK**.

In your Visual Basic code, use the `New` method to create the server:

```
MatLab = New MApp.MApp
```

View MATLAB Automation methods from the Visual Basic Object Browser under the Library called `MLAPP`.

Get or Set the Status of a MATLAB Automation Server

To make MATLAB an Automation server, use the `enableservice` function. For example, type:

```
enableservice('AutomationServer',true)
```

To determine the current state of a MATLAB Automation server, type:

```
enableservice('AutomationServer')
```

If MATLAB displays:

```
ans =  
    1
```

then MATLAB is currently an Automation server.

See Also

`enableservice`

Related Examples

- “Manually Create Automation Server” on page 18-13

More About

- “Register MATLAB as Automation Server” on page 18-2
- “Programmatic Identifiers” on page 18-4
- “MATLAB Startup Folder”

Connect to Existing MATLAB Server

It is not always necessary to create a new instance of a MATLAB server whenever your application needs some task done in MATLAB. Clients can connect to an existing MATLAB Automation server using the `actxGetRunningServer` function or by using a command similar to the Visual Basic .NET `GetObject` command.

Using Visual Basic .NET Code

The Visual Basic .NET command shown here returns a handle `h` to the MATLAB server application:

```
h = GetObject(, "matlab.application")
```

Note It is important to use the syntax shown above to connect to an existing MATLAB Automation server. Omit the first argument, and make sure the second argument is as shown.

The following Visual Basic .NET example connects to an existing MATLAB server, then executes a plot command in the server. If you do not already have a MATLAB server running, create one following the instructions in “Access Methods from the Visual Basic Object Browser” on page 18-7.

```
Dim h As Object
h = GetObject(, "matlab.application")

' Handle h should be valid now.
' Test it by calling Execute.
h.Execute ("plot([0 18], [7 23])")
```

VT_DATE Data Type

To pass a VT_DATE type input to a Visual Basic program, use the MATLAB class COM.date. For example:

```
d = COM.date(2005,12,21,15,30,05);  
get(d)  
    Value: 7.3267e+005  
    String: '12/21/2005 3:30:05 PM'
```

Use the now function to set the Value property to a date number:

```
d.Value = now;
```

COM.date accepts the same input arguments as datenum.

Data Types For Visual Basic .NET Clients

Data types for the arguments and return values of the server functions are expressed as Automation data types, which are language-independent types defined by the Automation protocol.

For example, `BSTR` is a wide-character string type defined as an Automation type, and is the same data format used by the Visual Basic language to store strings. Any COM-compliant controller should support these data types, although the details of how you declare and manipulate these are controller specific.

Visible Property

You have the option of making MATLAB visible or not by setting the `Visible` property. When visible, MATLAB appears on the desktop, enabling the user to interact with it. This might be useful for such purposes as debugging. When not visible, the MATLAB window does not appear, thus perhaps making for a cleaner interface and also preventing any interaction with the application.

By default, the `Visible` property is enabled, or set to `1`. You can change the `Visible` property by setting it to `0` (invisible) or `1` (visible).

Manually Create Automation Server

Microsoft Windows operating system automatically creates an Automation server when a controller application first establishes a server connection. Alternatively, you can manually create the server before starting any of the client processes.

To create a MATLAB server manually, use the `-automation` switch in the `matlab` startup command. From the operating system prompt, type:

```
matlab -automation
```

Alternatively, you can add this switch every time you run MATLAB, as follows:

- 1 Right-click the MATLAB shortcut icon



and select **Properties** from the context menu. The Properties dialog box for `matlab.exe` opens to the **Shortcut** tab.

- 2 In the **Target** field, add `-automation` to the end of the target path for `matlab.exe`. Be sure to include a space between the file name and the symbol `-`. For example:

```
"C:\Program Files\MATLAB\R2016a\bin\win64\MATLAB.exe" -automation
```

Note When the operating system automatically creates a MATLAB server, it too uses the `-automation` switch. In this way, MATLAB servers are differentiated from other MATLAB sessions. This protects controllers from interfering with any interactive MATLAB sessions that might be running.

Start MATLAB as Automation Server in Desktop Mode

To start MATLAB as a COM Automation server in full desktop mode, use the programmatic identifier `Matlab.Desktop.Application`. For example, in Microsoft Visual Basic .NET:

```
Dim MatLab As Object  
Dim Result As String  
MatLab = CreateObject("Matlab.Desktop.Application")  
Result = MatLab.Execute("surf(peaks)")
```


Call MATLAB Function from Visual Basic .NET Client

This example calls MATLAB functions from a Microsoft Visual Basic client application through a COM interface. The example plots a graph in a new MATLAB window and performs a simple computation.

```
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag(1, 3) As Double

MatLab = CreateObject("Matlab.Application")

'Call MATLAB function from VB
Result = MatLab.Execute("surf(peaks)")

'Execute simple computation
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8]")
Result = MatLab.Execute("b = a + a ")

'Bring matrix b into VB program
MatLab.GetFullMatrix("b", "base", MReal, MImag)
```

See Also

[Execute](#) | [GetFullMatrix](#)

More About

- “Calling MATLAB as COM Automation Server”

Pass Complex Data to MATLAB from C# Client

This example creates complex data in the client C# program and passes it to MATLAB. The matrix consists of a vector of real values in variable `pr` and of imaginary values in `pi`. The example reads the matrix back into the C# program.

The reference to the MATLAB Type Library for C# is:

```
MApp.MLApp matlab = new MApp.MLApp();
```

From your C# client program, add a reference to your project to the MATLAB COM object. For example, in Microsoft Visual Studio, open your project. From the **Project** menu, select **Add Reference**. Select the **COM** tab in the Add Reference dialog box. Select the MATLAB application. Refer to your vendor documentation for details.

Here is the complete example:

```
using System;
namespace ConsoleApplication4
{
class Class1
{
[STAThread]
static void Main(string[] args)
{
MApp.MLApp matlab = new MApp.MLApp();

System.Array pr = new double[4];
pr.SetValue(11,0);
pr.SetValue(12,1);
pr.SetValue(13,2);
pr.SetValue(14,3);

System.Array pi = new double[4];
pi.SetValue(1,0);
pi.SetValue(2,1);
pi.SetValue(3,2);
pi.SetValue(4,3);

matlab.PutFullMatrix("a", "base", pr, pi);

System.Array prresult = new double[4];
System.Array piresult = new double[4];
```

```
matlab.GetFullMatrix("a", "base", ref prresult, ref piresult);  
}  
}  
}
```

See Also

[GetFullMatrix](#) | [PutFullMatrix](#)

Call MATLAB Function from C# Client

This example shows how to call a user-defined MATLAB function `myfunc` from a C# application.

Create a MATLAB function, `myfunc`, in the folder `c:\temp\example`.

```
function [x,y] = myfunc(a,b,c)
x = a + b;
y = sprintf('Hello %s',c);
```

Create the C# application.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create the MATLAB instance
            MLab.MLab matlab = new MLab.MLab();

            // Change to the directory where the function is located
            matlab.Execute(@"cd c:\temp\example");

            // Define the output
            object result = null;

            // Call the MATLAB function myfunc
            matlab.Feval("myfunc", 2, out result, 3.14, 42.0, "world");

            // Display result
            object[] res = result as object[];

            Console.WriteLine(res[0]);
            Console.WriteLine(res[1]);
            Console.ReadLine();
        }
    }
}
```

From your C# client program, add a reference to your project to the MATLAB COM object. For example, in Microsoft Visual Studio, open your project. From the **Project** menu, select **Add Reference**. Select the **COM** tab in the Add Reference dialog box. Select the MATLAB application. Refer to your vendor documentation for details.

See Also

Execute | Feval (COM)

More About

- “Calling MATLAB as COM Automation Server”

View MATLAB Functions from Visual Basic .NET Object Browser

You can find out what methods are available from a MATLAB Automation server using the Object Browser of your Microsoft Visual Basic client application. To do this, follow this procedure in the client application to reference the MATLAB Application Type Library:

- 1 Select the **Project** menu.
- 2 Select **Reference** from the subsequent menu.
- 3 Check the box next to the **MATLAB Application Type Library**.
- 4 Click **OK**.

This enables you to view MATLAB Automation methods from the Visual Basic Object Browser under the Library called MLAPP. You can also see a list of MATLAB Automation methods when you use the term `Matlab` followed by a period. For example:

```
Dim Matlab As MApp.MApp
Private Sub View_Methods()
Matlab = New MApp.MApp
'The next line shows a list of MATLAB Automation methods
Matlab.
End Sub
```

Waiting for MATLAB Application to Complete

When you call a MATLAB function from another program, the program might display a timeout message while waiting for the MATLAB function to complete. Refer to solutions posted in MATLAB Answers for tips for handling alerts from other programming languages.

See Also

External Websites

- Why does Microsoft Excel generate the message "Microsoft Excel is waiting for another application to complete an OLE action." when I use Spreadsheet Link EX?

Conversion of MATLAB Types to COM Types

The following table shows how MATLAB converts data from MATLAB to COM types.

MATLAB Type	Closest COM Type	Allowed Types
handle	VT_DISPATCH VT_UNKNOWN	VT_DISPATCH VT_UNKNOWN
character vector	VT_BSTR	VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE
int16	VT_I2	VT_I2
uint16	VT_UI2	VT_UI2
int32	VT_I4	VT_I4 VT_INT
uint32	VT_UI4	VT_UI4 VT_UINT
int64	VT_I8	VT_I8
uint64	VT_UI8	VT_UI8
single	VT_R4	VT_R4
double	VT_R8	VT_R8 VT_CY
logical	VT_BOOL	VT_BOOL
char	VT_I1	VT_I1 VT_UI1

Variant Data

variant is any data type except a structure or a sparse array. (For more information, see “Fundamental MATLAB Classes”.)

When used as an input argument, MATLAB treats `variant` and `variant(pointer)` the same way.

If you pass an empty array (`[]`) of type `double`, MATLAB creates a `variant(pointer)` set to `VT_EMPTY`. Passing an empty array of any other numeric type is not supported.

MATLAB Argument	Closest COM Type	Allowed Types
<code>variant</code>	<code>VT_VARIANT</code>	<code>VT_VARIANT</code> <code>VT_USERDEFINED</code> <code>VT_ARRAY</code>
<code>variant(pointer)</code>	<code>VT_VARIANT</code>	<code>VT_VARIANT</code> <code>VT_BYREF</code>

SAFEARRAY Data

When a COM method identifies a `SAFEARRAY` or `SAFEARRAY(pointer)`, the MATLAB equivalent is a matrix.

MATLAB Argument	Closest COM Type	Allowed Types
<code>SAFEARRAY</code>	<code>VT_SAFEARRAY</code>	<code>VT_SAFEARRAY</code>
<code>SAFEARRAY(pointer)</code>	<code>VT_SAFEARRAY</code>	<code>VT_SAFEARRAY</code> <code>VT_BYREF</code>

Unsupported Types

MATLAB does not support the following COM types.

- Structure
- Sparse array
- Multidimensional `SAFEARRAY`s (greater than two dimensions)
- Write-only properties

See Also

`GetWorkspaceData`

Conversion of COM Types to MATLAB Types

The following table shows how MATLAB converts data from a COM application into MATLAB types.

COM Variant Type	Description	MATLAB Type
VT_DISPATCH	IDispatch *	handle
VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE	wide null terminated string null terminated string OLE Automation string FILETIME SCODE 16-byte fixed point Class ID date	character vector
VT_INT VT_UINT VT_I2 VT_UI2 VT_I4 VT_UI4 VT_R4 VT_R8 VT_CY	signed machine int unsigned machine int 2 byte signed int unsigned short 4 byte signed int unsigned long 4 byte real 8 byte real currency	double
VT_I8	signed int64	int64
VT_UI8	unsigned int64	uint64
VT_BOOL		logical
VT_I1 VT_UI1	signed char unsigned char	char
VT_VARIANT VT_USERDEFINED VT_ARRAY	VARIANT * user-defined type SAFEARRAY*	variant
VT_VARIANT VT_BYREF	VARIANT * void* for local use	variant(pointer)
VT_SAFEARRAY	use VT_ARRAY in VARIANT	SAFEARRAY

COM Variant Type	Description	MATLAB Type
VT_SAFEARRAY VT_BYREF		SAFEARRAY(pointer)

See Also

PutWorkspaceData

Using Web Services with MATLAB

- “Display Streamed Data in Figure Window” on page 19-2
- “Display JPEG Images Streamed from IP Camera” on page 19-8
- “Send Multipart Form Messages” on page 19-12
- “Send and Receive HTTP Messages” on page 19-14
- “What Is the HTTP Interface?” on page 19-18
- “Manage Cookies” on page 19-19
- “HTTP Data Type Conversion” on page 19-21
- “Display Progress Monitor for HTTP Message” on page 19-30
- “Set Up WSDL Tools” on page 19-34
- “Display a World Map” on page 19-35
- “Using WSDL Web Service with MATLAB” on page 19-40
- “Access Services That Use WSDL Documents” on page 19-43
- “Error Handling” on page 19-45
- “XML-MATLAB Data Type Conversion” on page 19-47
- “Limitations to WSDL Document Support” on page 19-49
- “Manually Redirect HTTP Messages” on page 19-53

Display Streamed Data in Figure Window

This example shows how to customize an HTTP `StringConsumer` class—`PricesStreamer`—to display streamed data from a hypothetical website in a MATLAB figure window. To create a working example:

- Identify a URL, similar to:

```
url = matlab.net.URI('<URL>', 'accountId', <YOUR_ACCOUNT_ID>, '<NAME>', '<VALUE>');
```

- Modify `PricesStreamer.putData` to read data specific to your web service

The following tasks are described in this topic. For information about displaying table data in a figure window, see `uitable`.

In this section...

“`PricesStreamer` Class” on page 19-2

“Map Data to MATLAB `uitable` Object” on page 19-5

“Display Data in JSON Format” on page 19-6

“Terminate Data Stream” on page 19-6

“Call `PricesStreamer`” on page 19-7

PricesStreamer Class

`PricesStreamer.m` is a subclass of the `StringConsumer` class, which is a subclass of `ContentConsumer`. `PricesStreamer` receives streamed data customized to the data provided by a specific web service. In this example, the structure of the data is:

```
% Data contains one or more CRLF-separated JSON structures.
% Each structure is one of the format:
% {"heartbeat"="timestamp"}
% {"tick"="timestamp", "bid"=bid, "ask"=ask}
% where timestamp is a GMT time and bid and ask are numbers.
```

MATLAB calls the `PricesStreamer.putData` function for each chunk of data received from the server. The function first converts the raw `uint8` bytes to a JSON string using `StringConsumer`. Next, it gets a MATLAB structure from the JSON string using `jsondecode` and then displays the data in a table in a figure, adding one line to the top of the table for each increment of data. You can modify the `putData` function to do something else with the data, for example, plot a real-time graph or display delta prices.

PricesStreamer sets the stop return value to stop the operation when the user closes the figure. For more information, see putData.

```

classdef PricesStreamer < matlab.net.http.io.StringConsumer
    % PricesStreamer accepts streamed JSON
    % and displays the result in a uitable in a figure window.

    % Copyright 2016-2017 The MathWorks, Inc.
    properties
        Figure
        Table
        Endit logical
        HaveTick logical
    end

    methods (Access=protected)
        function length = start(obj)
            if obj.Response.StatusCode ~= matlab.net.http.StatusCode.OK
                length = 0;
            else
                length = obj.start@matlab.net.http.io.StringConsumer;
                obj.Figure = figure('CloseRequestFcn',@obj.endit);
                obj.Figure.Position(4) = 550;
                obj.Figure.Position(2) = 50;
                obj.Table = uitable(obj.Figure,...
                    'ColumnName',{'Time','Bid','Ask'},...
                    'ColumnWidth',{130,'auto','auto'});
                obj.Table.Position(4) = 500;
                obj.Table.Data = cell(0,3);
                obj.Endit = false;
                obj.HaveTick = false;
            end
        end
    end

    methods
        function [len,stop] = putData(obj, data)
            % Data contains one or more CRLF-separated JSON structures.
            % Each structure is one of the format:
            % {"heartbeat"="timestamp"}
            % {"tick"="timestamp", "bid"=bid, "ask"=ask}
            % where timestamp is a GMT time and bid and ask are numbers.
            if obj.Endit
                data = [];
            end
        end
    end
end

```

```
        delete(obj.Figure);
    end
    first = obj.CurrentLength + 1;
    [len,stop] = obj.putData@matlab.net.http.io.StringConsumer(data);
    if isempty(data) || stop
        if ischar(data) % data == '' means user ctrl/c'ed, so set to
            obj.Endit = true; % delete figure on next close
        end
        stop = true;
    else
        stop = false;
        last = obj.CurrentLength;
        newData = obj.Response.Body.Data.extractBetween(first,last);
        % split at CRLFs
        strings = strsplit(newData, '\r\n');
        try
            cellfun(@obj.displayJSON, strings);
        catch e
            fprintf('Error on JSON:\n%s<EOF>\n',data);
            obj.Endit = true;
            rethrow(e);
        end
    end
end

function displayJSON(obj, str)
    if ~isempty(str)
        try
            val = jsondecode(str);
        catch e
            fprintf('Error "%s" on JSON:\n%s<EOF>\n',e.message,str);
            rethrow(e);
        end
        if isfield(val,'tick')
            tick = val.tick;
            newdata = {cvtime(val.tick.time),tick.bid,tick.ask};
            setExtent = ~obj.HaveTick;
            obj.HaveTick = true;
        elseif isfield(val, 'heartbeat')
            newdata = {cvtime(val.heartbeat.time),'',''};
            setExtent = false;
        end
        obj.Table.Data = [newdata;obj.Table.Data];
        if setExtent || ~mod(log10(length(obj.Table.Data)),1)
```



```

        % set extent on first tick and every power of 10
        % add 15 for width of scroll bar
        obj.Table.Position(3) = obj.Table.Extent(3) + 15;
    end
    drawnow
end
end

function endit(obj,~,~)
    % endit callback from close(obj.Figure)
    if exist('obj','var') && isvalid(obj)
        if obj.Endit
            if isvalid(obj.Figure)
                delete(obj.Figure);
            end
        else
            obj.Endit = true;
        end
    end
end

function delete(obj)
    if ~isempty(obj.Figure) && isvalid(obj.Figure)
        delete(obj.Figure);
    end
end

end

function time = cvtime(time)
    % Format time data for display
    time = datetime(time,'InputFormat','yyyy-MM-dd'T'HH:mm:ss.S'Z','TimeZone','GMT');
    time.TimeZone = 'local';
    time = char(time, 'dd-MMM-yyyy HH:mm:ss.S');
end

```

Map Data to MATLAB uitable Object

Identify the data structures for your use case by reading API information from the web service. The data for this example contains one or more CRLF-separated JSON structures. The format for the structures is one of the following, where `timestamp` is a GMT time and `bid` and `ask` are numbers.

- {"heartbeat"="timestamp"}
- {"tick"="timestamp", "bid"=bid, "ask"=ask}

To read this specific format, override the `putData` method. The following statements from the `PricesStreamer` class use `StringConsumer.putData` to read the next buffer, then select the JSON strings.

```
first = obj.CurrentLength + 1;
[len,stop] = obj.putData@matlab.net.http.io.StringConsumer(data);
last = obj.CurrentLength;
newData = obj.Response.Body.Data.extractBetween(first,last);
% split at CRLFs
strings = strsplit(newData, '\r\n');
```

Display Data in JSON Format

The following statements from the `displayJSON` function individually handle the JSON tick and heartbeat structures. A helper function `cvtime` formats the time data for display in the table.

```
function displayJSON(obj, str)
...
val = jsondecode(str);
if isfield(val, 'tick')
    tick = val.tick;
    newdata = {cvtime(val.tick.time), tick.bid, tick.ask};
...
elseif isfield(val, 'heartbeat')
    newdata = {cvtime(val.heartbeat.time), '', ''};
...
end
obj.Table.Data = [newdata;obj.Table.Data];
...
end
```

Terminate Data Stream

In this example, MATLAB receives data as long as the web service is active. The user can terminate the stream by closing the figure window or by pressing **Ctrl+C**. To inform MATLAB of a user interruption, set the `stop` argument in `putData` to `false`. Clean-up tasks include closing the figure using the `CloseRequestFcn` property and deleting the object using the `PricesStreamer.delete` function.

Call PricesStreamer

The following code provides a framework for retrieving data from a web service. To run the code, you must provide values for content within <> characters. The URL for your web service might include additional parameters, such as login information and other information specified as name, value pair arguments. To utilize the PricesStreamer, add it to your call to send. For information about creating request messages, see “HTTP Interface”.

```
url = matlab.net.URI('<URL>', 'accountId', <YOUR_ACCOUNT_ID>, '<NAME>', '<VALUE>');
authInfo = matlab.net.http.AuthInfo(matlab.net.http.AuthenticationScheme.Bearer, ...
    'Encoded', '<YOUR_CREDENTIALS>');
af = matlab.net.http.field.AuthorizationField('Authorization', authInfo);
r = matlab.net.http.RequestMessage('get', af);
consumer = PricesStreamer;
% SavePayload set to retain all results - useful for debugging
[resp, req, hist] = r.send(url, matlab.net.http.HTTPOptions('SavePayload', true), consumer);
% Show the results for debugging
show(resp)
```

The following is an example of data from a web service sending data described in “Map Data to MATLABuitable Object” on page 19-5.

```
HTTP/1.1 200 Ok
Server: openresty/1.9.15.1
Date: Wed, 06 Sep 2017 19:26:56 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Connection: close
Access-Control-Allow-Origin: *

{"tick":{"instrument":"AUD_CAD","time":"2017-09-06T19:26:54.304054Z","bid":0.97679,"ask":0.97703}}
{"heartbeat":{"time":"2017-09-06T19:26:56.253091Z"}}
{"tick":{"instrument":"AUD_CAD","time":"2017-09-06T19:26:57.226918Z","bid":0.97678,"ask":0.97703}}
{"tick":{"instrument":"AUD_CAD","time":"2017-09-06T19:26:58.226909Z","bid":0.97678,"ask":0.97705}}
{"heartbeat":{"time":"2017-09-06T19:26:58.720409Z"}}
{"tick":{"instrument":"AUD_CAD","time":"2017-09-06T19:27:00.733194Z","bid":0.97679,"ask":0.97704}}
{"heartbeat":{"time":"2017-09-06T19:27:01.251202Z"}}
{"tick":{"instrument":"AUD_CAD","time":"2017-09-06T19:27:01.757501Z","bid":0.9768,"ask":0.97706}}
{"heartbeat":{"time":"2017-09-06T19:27:03.720469Z"}}
```

See Also

[matlab.net.http.io.StringConsumer](#) | [uitable](#)

Display JPEG Images Streamed from IP Camera

This example shows how to use an HTTP MultipartConsumer to stream video from a website. It customizes an ImageConsumer class-CameraPlayer-to display JPEG images from a hypothetical website derived from an IP address.

To create a working example, you need:

- IP address-based URL similar to:

```
url = 'http://999.99.99.99/video/mjpg.cgi';
```

- Credentials similar to:

```
creds = matlab.net.http.Credentials('Scheme','Basic','User','admin','Pass','admin');  
creds.Username = 'yourName';  
creds.Password = 'yourPassword';
```

CameraPlayer Class

An IP camera sends JPEG images as parts of a never-ending multipart message. Each part has a type of "image/jpeg". To process this message, create a MultipartConsumer indicating that any part whose type is "image/*" should be handled by CameraPlayer. MATLAB calls the CameraPlayer.putData method for each frame of the image received, which calls its superclass to convert the data. CameraPlayer in turn uses imshow to create a figure window to display the image, and displays subsequent images in the same window. When the user closes the window, putData returns stop=true, which causes MATLAB to close the connection.

```
classdef CameraPlayer < matlab.net.http.io.ImageConsumer  
% CameraPlayer Player for IP camera  
% A ContentConsumer that displays image content types in a figure window. If  
% specified directly in the RequestMessage.send() operation, it assumes the  
% entire contents of the ResponseMessage is a single image. If the response  
% is a multipart message, and this is specified as a handler for image  
% types to a GenericConsumer or MultipartConsumer, then this assumes each  
% part is a frame of a video stream and displays them as they are received.  
%  
% CameraPlayer properties:  
% CameraPlayer - constructor  
% Image - the currently displayed Image object  
%  
% The following displays a received image, or a sequence of images received
```

```

% in a multipart message and saves the last image received in the response
% message Data. The last image displayed remains visible until cp is
% deleted.
%
% req = RequestMessage;
% cp = CameraPlayer;
% resp = req.send(url, [], GenericConsumer('image/*', cp));
% image = cp.Image;
% ...operate on image data...

```

```

% Copyright 2017 The MathWorks, Inc.

```

properties

```

% Image - the currently displayed Image object
% This image is in an Axes in a Figure. It is deleted when this object is
% deleted.
Image
% Enough - control number of times to display message
Enough

```

```

end

```

methods

```

function obj = CameraPlayer()
    obj = obj@matlab.net.http.io.ImageConsumer();
end

```

```

function [len, stop] = putData(obj, data)
% putData - called by MATLAB or parent Consumer to process image data

```

```

% Pass the data to ImageConsumer, which buffers it until the end of data
% and then converts it to a MATLAB image depending on the type of image.
[len, stop] = obj.putData@matlab.net.http.io.ImageConsumer(data);
if isempty(data)
    % end of image; display the result in Response.Body.Data
    imdata = obj.Response.Body.Data;
    if iscell(imdata)
        if ~isempty(imdata{2})
            % If it was an indexed image if we get a cell array, so convert
            imdata = ind2rgb(imdata{1},imdata{2});
        else
            imdata = imdata{1};
        end
    end
    if isempty(obj.Image)

```

```
        % First time we are called, create a figure containing the image
        obj.Image = imshow(imdata);
        obj.Enough = 0;
    else
        % Subsequent times we are called, just change CData in an already-
        % image.

        obj.Enough = obj.Enough+1;
        if obj.Enough > 100
            disp 'To stop, close figure window.'
            obj.Enough = 0;
        end
        try
            obj.Image.CData = imdata;
        catch e
            % Fails if window closed or data was bad
            if strcmp(e.identifier, 'MATLAB:class:InvalidHandle')
                % User must have closed the window; terminate silently
                stop = true;
                disp 'Figure window closed.'
                return
            end
        end
        end
        drawnow
    end
end

function delete(obj)
    delete(obj.Image);
end
end
end
```

Call CameraPlayer

The following code provides a framework for retrieving images. To run the code, you must provide values for content within <> characters. The URL for your web service might include additional parameters, such as login information and other information specified as name, value pair arguments. To utilize the CameraPlayer, add it to your call to send. For information about creating request messages, see “HTTP Interface”.

```
url = '<YOUR_URL_CONTAINING_IP_ADDRESS>';
cp = CameraPlayer;
```

```
consumer = matlab.net.http.io.MultipartConsumer('image/*',cp);
creds = matlab.net.http.Credentials('Scheme','Basic','User','admin','Pass','admin');
creds.Username = '<YOURNAME>';
creds.Password = '<YOURPASSWORD>';
opts = matlab.net.http.HTTPOptions('Cred',creds,'SavePayload',true);
r = matlab.net.http.RequestMessage();
resp = r.send(url, opts, consumer);
```

See Also

Send Multipart Form Messages

The following code sends one file in a PUT message.

```
import matlab.net.http.*
import matlab.net.http.field.*
import matlab.net.http.io.*
provider = FileProvider('dir/imageFile.jpg');
req = RequestMessage(PUT,[],provider);
resp = req.send(url);
```

The provider sets the appropriate Content-Type header field in the request message to the type of file derived from the file name extension, and adds a Content-Disposition field naming the file. In this example, the values are "image/jpeg" with file name "imageFile.jpg".

To upload multiple files in a "multipart/mixed" message, possibly of different types, create an array of FileProviders by specifying an array of file names and use this array as a delegate to a MultipartProvider. In the following code, each header of the multipart message contains a Content-Type field and a Content-Disposition field with a file name attribute naming the file.

```
url = "www.somewebsite.com";
provider = MultipartProvider(FileProvider(["image1.jpg", "file1.txt"]));
req = RequestMessage(PUT, [], provider);
resp = req.send(url);
```

In practice, most servers that accept multipart content expect it to be of type "multipart/form-data", not "multipart/mixed". To send files using multipart forms, use a MultipartFormProvider. That provider requires you to know the control names for the various fields of the form, so that each part is associated with the correct control. For example, to send a form with controls called "files" and "text", where the first accepts multiple files and the second accepts just one, create a provider based on the following code:

```
provider = MultipartFormProvider("files", FileProvider(["image1.jpg", "file1.txt"]),...
    "text", FileProvider("file1.txt"));
```

If a server requires multiple files specified by using a nested format, use the following code pattern:

```
provider = MultipartFormProvider("files", MultipartProvider(FileProvider(["image1.jpg", "file1.txt"]),...
    "text", FileProvider("file1.txt"));
```


See Also

Send and Receive HTTP Messages

This example shows how to send a request to a server that involves redirection and might require digest authentication.

The `sendRequest` function automatically redirects and authenticates on the first request. For subsequent requests, you do not want to pay the cost of a redirect. To do this, `sendRequest` saves the cookies received on previous requests and reuses them for subsequent requests.

`sendRequest` illustrates the use of the history, the reuse of a custom `HTTPOptions` object to change the default timeout, and the use of saved credentials.

This example does not illustrate a robust, general-purpose mechanism for achieving these results. In particular, `sendRequest` returns all cookies received from a host in every subsequent message to that host, regardless of the domain, path, expiration, or other properties. `sendRequest` also does not save credentials or cookies across MATLAB sessions. Nonetheless, `sendRequest` is adequate for many applications.

Create the `sendRequest` function from the following code.

```
function response = sendRequest(uri,request)

% uri: matlab.net.URI
% request: matlab.net.http.RequestMessage
% response: matlab.net.http.ResponseMessage

% matlab.net.http.HTTPOptions persists across requests to reuse previous
% Credentials in it for subsequent authentications
persistent options

% infos is a containers.Map object where:
%   key is uri.Host;
%   value is "info" struct containing:
%       cookies: vector of matlab.net.http.Cookie or empty
%       uri: target matlab.net.URI if redirect, or empty
persistent infos

if isempty(options)
    options = matlab.net.http.HTTPOptions('ConnectTimeout',20);
end

if isempty(infos)
```

```

        infos = containers.Map;
end
host = string(uri.Host); % get Host from URI
try
    % get info struct for host in map
    info = infos(host);
    if ~isempty(info.uri)
        % If it has a uri field, it means a redirect previously
        % took place, so replace requested URI with redirect URI.
        uri = info.uri;
    end
    if ~isempty(info.cookies)
        % If it has cookies, it means we previously received cookies from this host.
        % Add Cookie header field containing all of them.
        request = request.addFields(matlab.net.http.field.CookieField(info.cookies));
    end
catch
    % no previous redirect or cookies for this host
    info = [];
end

% Send request and get response and history of transaction.
[response, ~, history] = request.send(uri, options);
if response.StatusCode ~= matlab.net.http.StatusCode.OK
    return
end

% Get the Set-Cookie header fields from response message in
% each history record and save them in the map.
arrayfun(@addCookies, history)

% If the last URI in the history is different from the URI sent in the original
% request, then this was a redirect. Save the new target URI in the host info struct.
targetURI = history(end).URI;
if ~isequal(targetURI, uri)
    if isempty(info)
        % no previous info for this host in map, create new one
        infos(char(host)) = struct('cookies', [], 'uri', targetURI);
    else
        % change URI in info for this host and put it back in map
        info.uri = targetURI;
        infos(char(host)) = info;
    end
end
end

```

```
function addCookies(record)
    % Add cookies in Response message in history record
    % to the map entry for the host to which the request was directed.
    %
    ahost = record.URI.Host; % the host the request was sent to
    cookieFields = record.Response.getFields('Set-Cookie');
    if isempty(cookieFields)
        return
    end
    cookieData = cookieFields.convert(); % get array of Set-Cookie structs
    cookies = [cookieData.Cookie]; % get array of Cookies from all structs
    try
        % If info for this host was already in the map, add its cookies to it.
        ainfo = infos(ahost);
        ainfo.cookies = [ainfo.cookies cookies];
        infos(char(ahost)) = ainfo;
    catch
        % Not yet in map, so add new info struct.
        infos(char(ahost)) = struct('cookies',cookies,'uri',[]);
    end
end
end
```

Call the function.

```
request = matlab.net.http.RequestMessage;
uri = matlab.net.URI('https://www.mathworks.com/products');
response = sendRequest(uri,request)
```

response = ResponseMessage with properties:

```
StatusLine: 'HTTP/1.1 200 OK'
StatusCode: OK
Header: [1x11 matlab.net.http.HeaderField]
Body: [1x1 matlab.net.http.MessageBody]
Completed: 0
```

Your response values might be different.

See Also

[CookieField](#) | [HTTPOptions](#) | [LogRecord](#) | [ResponseMessage](#) | [URI](#) | [addFields](#) | [send](#)

What Is the HTTP Interface?

The MATLAB HTTP Interface provides functionality to issue properly structured HTTP requests and process their responses. You can also use the RESTful web services functions, `webread` and `webwrite`, to make HTTP requests. However, some interactions with a web service are more complex and require functionality not supported by these functions.

The HTTP interface is designed for programmatic use in scripts and functions. The interface is an object model for HTTP data structures. It includes classes for messages, their headers and fields, and other entities defined in the Internet Engineering Task Force (IETF®) standards. For more information, see the Request for Comments (RFC) documents RFC 7230, RFC 7231, RFC 7235, RFC 2617 and RFC 6265. The interface contains functions that implement semantics of HTTP messaging and utilities for processing the data sent and received. It also contains support classes required to process, transmit, and receive messages.

The interface is based on the HTTP/1.1 protocol.

See Also

More About

- “Web Access”

External Websites

- The Internet Engineering Task Force (IETF®)

Manage Cookies

The HTTP interface provides classes for you to manage cookies in HTTP messages. For information about cookies, see RFC 6265 HTTP State Management Mechanism.

The `Cookie` class contains the name and value of the cookie. For example, display the properties of a `Cookie` object `C`.

```
C =  
Cookie with properties:  
  
Name: "JSESSIONID"  
Value: "688412d8ed1cdf15f4a736dc6ab3"
```

The `CookieInfo` class contains the cookie and its attributes. The attributes are the properties of the `CookieInfo` object. For example, `info` is a `CookieInfo` object containing cookie `C`.

```
class(info)  
  
ans = matlab.net.http.CookieInfo
```

Display attributes of the cookie.

```
info.Expires  
  
ans =  
Sun, 26 Mar 2084 17:15:30 GMT  
  
info.CreationTime  
  
ans =  
Tue, 07 Jun 2016 09:38:56 GMT
```

A server returns cookies in a response message using a `SetCookieField` object. To get a `CookieInfo` object, call the `SetCookieField.convert` method.

To send a cookie to the server, add a `CookieField` object to a request message.

See Also

[Cookie](#) | [CookieField](#) | [CookieInfo](#) | [SetCookieField](#)

External Websites

- RFC 6265 HTTP State Management Mechanism

HTTP Data Type Conversion

In this section...

“Converting Data in Request Messages” on page 19-21

“Converting Data in Response Messages” on page 19-26

“Supported Image Data Subtypes” on page 19-28

The MATLAB HTTP interface automatically converts data types used in HTTP messages to and from MATLAB types.

Converting Data in Request Messages

When sending a message with a payload, assign your MATLAB data to the `Data` property in a `MessageBody` object, then send it as a `Body` property in a `RequestMessage` object. The type of your MATLAB data depends on the HTTP Content-Type of the message. If you do not specify a Content-Type, then MATLAB assumes Content-Type values, as described in “Content-Type Not Specified” on page 19-24.

This table shows how MATLAB converts `Data` to a payload in a request message based on the type/subtype properties and the charset attribute that you specify in the Content-Type header field. The asterisk character (*) means any subtype.

Content-Type	MATLAB Type in <code>MessageBody.Data</code> Property
<code>application/json</code>	<p><code>Data</code> converted to a Unicode value using the <code>jsonencode</code> function. MATLAB then uses the <code>unicode2native</code> function to convert the value to <code>uint8</code>, based on the charset attribute in the Content-Type header field.</p> <p>If you already have JSON-encoded text, then assign the text to the <code>Payload</code> property instead of the <code>Data</code> property. MATLAB converts the value to <code>uint8</code> using the charset attribute.</p> <p>If the charset attribute is not specified, then the default charset value is UTF-8.</p>

Content-Type	MATLAB Type in <code>MessageBody.Data</code> Property
<p><code>text/*</code> for any subtype other than <code>csv</code> or <code>xml</code></p>	<p>If <code>Data</code> is a character or string array or a cell array of character vectors, MATLAB reshapes and concatenates the text by row to form a vector.</p> <p>If <code>Data</code> is any other type, MATLAB converts <code>Data</code> using the <code>string</code> function. The resulting string is converted to <code>uint8</code> based on the charset.</p> <p>If you did not specify a charset, the default depends on the subtype. For the following subtypes, the default is UTF-8:</p> <ul style="list-style-type: none"> • <code>json</code> • <code>html</code> • <code>javascript</code> • <code>css</code> • <code>calendar</code> <p>For all other subtypes, MATLAB determines the charset. If all characters are in the ASCII range, then the charset is US-ASCII. Otherwise, the charset is UTF-8.</p> <hr/> <p>Note Servers might not properly interpret text types encoded as UTF-8 without an explicit UTF-8 charset. For best results, explicitly specify UTF-8 if your data contains non-ASCII characters.</p>

Content-Type	MATLAB Type in <code>MessageBody.Data</code> Property
<p><code>image/*</code></p>	<p>Data must be image data in a form acceptable to the <code>imwrite</code> function. Conversion of Data to <code>uint8</code> depends on the subtype. For information about supported types and for controlling the conversion, see “Supported Image Data Subtypes” on page 19-28.</p> <p>To control the conversion of your image data or to override the type of conversion based on the subtype, specify additional arguments to <code>imwrite</code> using a cell array. If you specify an image format argument (<code>fmt</code>), then it overrides the default conversion.</p> <p>For example, the following code converts <code>imageData</code> to JPEG with compression quality 50 and sends the data to the specified url with Content-Type set to "image/jpeg".</p> <pre>body = MessageBody({imageData, 'jpg', 'Quality', 50}); req = RequestMessage('put', ContentTypeField('image/jpeg'), body); resp = req.send(url);</pre>
<ul style="list-style-type: none"> • <code>application/xml</code> • <code>text/xml</code> 	<p>If Data is an XML DOM in the form of a Java <code>org.w3c.dom.Document</code> object, MATLAB converts it using the <code>xmlwrite</code> function.</p> <p>If Data is a string or character vector, MATLAB converts it to <code>uint8</code> using the specified charset. If not specified, the default charset value is UTF-8.</p>
<p><code>application/x-www-form-urlencoded</code></p>	<p>If Data is a vector of <code>matlab.net.QueryParameter</code> objects, then MATLAB converts it to a URL-encoded string. If it is a string or character vector, it is left unchanged.</p>

Content-Type	MATLAB Type in <code>MessageBody.Data</code> Property
<p>audio/*</p>	<p>Data must be audio data in a form acceptable to the <code>audiowrite</code> function. Create a cell array containing an m-by-n matrix of audio data and a sampling rate in Hz. You can specify additional arguments to <code>audiowrite</code> by adding name-value pair arguments to the cell array.</p> <p>MATLAB supports the following audio types:</p> <ul style="list-style-type: none"> • <code>audio/x-wav</code> • <code>audio/wav</code> • <code>audio/mp4</code> • <code>audio/vnd.wav</code> • <code>application/ogg</code> • <code>audio/flac</code>
<ul style="list-style-type: none"> • <code>application/csv</code> • <code>text/csv</code> • <code>application/vnd.openxmlformats-officedocument.spreadsheetml.sheet</code> • <code>application/vnd.ms-excel</code> 	<p>Data must be a table in a form suitable for the <code>writetable</code> function.</p> <p>For <code>csv</code> subtypes, MATLAB converts Data to comma-delimited text using the specified charset. The default charset is US-ASCII.</p> <p>For the other types, MATLAB converts Data to Excel spreadsheet data.</p> <p>To specify additional name-value pair arguments to <code>writetable</code>, create a cell array containing Data and the additional arguments. If you specify a 'FileType' argument, that type must be consistent with the subtype you specify.</p>

Content-Type Not Specified

If you do not specify a Content-Type field in the request message, MATLAB assigns the type, subtype, and charset based on the type of the Data property. This assumed behavior might not result in the Content-Type you intended, or might fail to determine the type, so for best results, specify the Content-Type. The following table describes the assumed

Content-Type based on Data. Types not listed might be handled, but the behavior for unlisted types is not guaranteed to remain the same in future releases.

MessageBody.Data Property Type Content-Type Not Specified	Resulting Content-Type
string character array cell array of character vectors	text/plain
table	text/csv
cell vector whose first element is a table	text/csv — If FileType is csv, then there is a name,value pair in the vector with the value 'FileType', 'csv' or there is no such pair. application/vnd.openxmlformats-officedocument.spreadsheetml.sheet — If FileType is spreadsheet.
org.w3c.dom.Document	application/xml
uint8 vector	To send a uint8 vector without conversion and ignoring the Content-Type header field, set the Payload property instead of Data. To send character-based data with no conversion, use the unicode2native function. This function uses the charset attribute to convert Data to a uint8 vector.

If the type is not one of those listed in the table, then MATLAB determines whether it is one of the following character-based types:

- text/*
- any type with a charset
- application/*javascript
- application/vnd.wolfram.mathematica.package

MATLAB converts these types to a string, using the charset, if specified, or US-ASCII for text/plain, UTF-8 for the application types, and the default MATLAB encoding for the other types.

Converting Data in Response Messages

When receiving a message with a payload, MATLAB converts the incoming byte stream (`MessageBody.Data` property) to an appropriate MATLAB type.

The following table is a list of the Content-Types that MATLAB recognizes in a response message, based the type/subtype properties and the charset attribute in the received Content-Type field. MATLAB converts the data only if the `HTTPOptions.ConvertResponse` property is true, which is the default. In the table, the asterisk character (*) means any characters.

Response Message Content-Type	MATLAB Type in <code>MessageBody.Data</code> Property
<code>application/json</code>	Data is converted to a string based on the charset and then to MATLAB data using the <code>jsondecode</code> function.
<code>image/*</code>	Data is converted to an image using the <code>imread</code> function with the specified subtype as the format, using default arguments. If <code>imread</code> returns more than one value, then Data is a cell array. For the supported types of image data, see “Supported Image Data Subtypes” on page 19-28. If the subtype is not in this list, then the subtype is passed to <code>imwrite</code> as the format, which might or might not be supported.

Response Message Content-Type	MATLAB Type in <code>MessageBody.Data</code> Property
audio/*	<p>Data is converted using the <code>audioread</code> function to a cell array of two values, an m-by-n matrix of audio data and a sampling rate in Hz. The subtype determines the format used by <code>audioread</code>. The supported types are:</p> <ul style="list-style-type: none"> • audio/wav • audio/x-wav • audio/vnd.wav • audio/mp4 • audio/flac <p>application/ogg is not converted because ogg data does not necessarily contain audio only.</p>
text/csv text/comma-separated-values application/csv application/comma-separated-values	Data is converted to a table using <code>readtable</code> , with assumed 'FileType' of csv and charset, if specified, or the MATLAB default encoding.
application/*spreadsheet*	Data is converted to a table using <code>readtable</code> , with 'FileType' assumed to be 'spreadsheet'.
text/xml application/xml	<p>If Java is available, Data is converted to a Java <code>org.w3c.dom.Document</code> using the <code>xmlread</code> function.</p> <p>If Java is not available, Data is processed as <code>text/plain</code> with the UTF-8 charset.</p>

If the type is not one of those listed in the table, then MATLAB determines whether it is one of the following character-based types:

- text/*
- any type with a charset

- `application/*javascript`
- `application/vnd.wolfram.mathematica.package`

MATLAB converts these types to a string, using the charset, if specified, or US-ASCII for `text/plain`, UTF-8 for the application types, and the default MATLAB encoding for the other types.

If MATLAB does not support the type, or if the `HTTPOptions.ConvertResponse` property is set to `false`, then:

- If the type is character-based, then `Data` contains the payload converted to string.
- Otherwise, `Data` contains the raw `uint8` vector.

If conversion of incoming data is attempted but fails (for example, "image/jpeg" data is not valid JPEG data), then the `History` property in the `HTTPException` thrown by the `RequestMessage.send` method contains the `ResponseMessage` with the `Payload` property set to the `uint8` payload and, if the type is character-based, then `Data` is set to the payload converted to a string.

Supported Image Data Subtypes

The following subtypes are supported by the `imwrite` function as the specified format. For example, the format argument for subtype `bmp` is `'bmp'`. The `imread` function converts the data with the specified subtype as the format.

Subtype	Format Used by <code>imwrite</code> and <code>imread</code>
<code>bmp</code>	<code>'bmp'</code>
<code>gif</code>	<code>'gif'</code>
<code>jpeg</code>	<code>'jpeg'</code>
<code>jp2</code>	<code>'jp2'</code>
<code>jpx</code>	<code>'jpx'</code>
<code>png</code>	<code>'png'</code>
<code>tiff</code>	<code>'tiff'</code>
<code>x-hdf</code>	<code>'hdf'</code>
<code>x-portable-bitmap</code>	<code>'pbm'</code>

Subtype	Format Used by imwrite and imread
x-pcx	'pcx'
x-portable-graymap	'pgm'
x-portable-anymap	'pnm'
x-portable-pixmap	'ppm'
x-cmu-raster	'ras'
x-xwd	'xwd'

See Also

ContentTypeField | HTTPException | HTTPOptions | MessageBody |
RequestMessage | ResponseMessage | audioread | audiowrite | imread | imwrite
| jsondecode | jsonencode | xmlread | xmlwrite

Display Progress Monitor for HTTP Message

This example shows how to implement a progress monitor, `MyProgressMonitor`, that displays a progress bar for data transferred to and from a website. The monitor displays a progress bar in a window created by the MATLAB `waitbar` function. It uses `set.Direction` and `set.Value` methods to monitor changes to the `Direction` and `Value` properties.

Each time MATLAB sets the `Direction` property, `MyProgressMonitor` creates a progress bar window and displays either a sending or a receiving message.

Create the following `MyProgressMonitor` class file.

The class initializes the `Interval` property to `.001` seconds because the example sends only 1 MB of data. The small interval allows you to observe the progress bar.

```
classdef MyProgressMonitor < matlab.net.http.ProgressMonitor
    properties
        ProgHandle
        Direction matlab.net.http.MessageType
        Value uint64
        NewDir matlab.net.http.MessageType = matlab.net.http.MessageType.Request
    end

    methods
        function obj = MyProgressMonitor
            obj.Interval = .001;
        end

        function done(obj)
            obj.closeit();
        end

        function delete(obj)
            obj.closeit();
        end

        function set.Direction(obj, dir)
            obj.Direction = dir;
            obj.changeDir();
        end

        function set.Value(obj, value)
```

```

        obj.Value = value;
        obj.update();
    end
end

methods (Access = private)
function update(obj,~)
    % called when Value is set
    import matlab.net.http.*
    if ~isempty(obj.Value)
        if isempty(obj.Max)
            % no maximum means we don't know length, so message
            % changes on every call
            value = 0;
            if obj.Direction == MessageType.Request
                msg = sprintf('Sent %d bytes...', obj.Value);
            else
                msg = sprintf('Received %d bytes...', obj.Value);
            end
        else
            % maximum known, update proportional value
            value = double(obj.Value)/double(obj.Max);
            if obj.NewDir == MessageType.Request
                % message changes only on change of direction
                if obj.Direction == MessageType.Request
                    msg = 'Sending...';
                else
                    msg = 'Receiving...';
                end
            end
        end
    end
    if isempty(obj.ProgHandle)
        % if we don't have a progress bar, display it for first time
        obj.ProgHandle = ...
            waitbar(value, msg, 'CreateCancelBtn', ...
                @(~,~)cancelAndClose(obj));
        obj.NewDir = MessageType.Response;
    elseif obj.NewDir == MessageType.Request || isempty(obj.Max)
        % on change of direction or if no maximum known, change message
        waitbar(value, obj.ProgHandle, msg);
        obj.NewDir = MessageType.Response;
    else
        % no direction change else just update proportional value
        waitbar(value, obj.ProgHandle);
    end
end
end

```

```
        end
    end

    function cancelAndClose(obj)
        % Call the required CancelFcn and then close our progress bar.
        % This is called when user clicks cancel or closes the window.
        obj.CancelFcn();
        obj.closeit();
    end
end

function changeDir(obj,~)
    % Called when Direction is set or changed. Leave the progress
    % bar displayed.
    obj.NewDir = matlab.net.http.MessageType.Request;
end
end

methods (Access=private)
    function closeit(obj)
        % Close the progress bar by deleting the handle so
        % CloseRequestFcn isn't called, because waitbar calls
        % cancelAndClose(), which would cause recursion.
        if ~isempty(obj.ProgHandle)
            delete(obj.ProgHandle);
            obj.ProgHandle = [];
        end
    end
end
end
end
```

To start the operation, specify the progress monitor.

```
opt = matlab.net.http.HTTPOptions(...
    'ProgressMonitorFcn',@MyProgressMonitor,...
    'UseProgressMonitor',true);
```

Create the data.

```
x = ones(1000000,1,'uint8');
body = matlab.net.http.MessageBody(x);
```

Create the message. The `httpbin.org/put` service returns data received in a PUT message.

```
url = matlab.net.URI('http://httpbin.org/put');  
method = matlab.net.http.RequestMethod.PUT;  
req = matlab.net.http.RequestMessage(method, [], body);
```

Send the message.

```
[resp,~,hist] = req.send(url,opt);
```

See Also

ProgressMonitor | waitfor

Set Up WSDL Tools

This example shows how to find information to install the programs required to use a WSDL web service in MATLAB. You need supported versions of the OpenJDK™ or Oracle® Java JDK and the Apache™ CXF programs. These programs require several hundred megabytes of disk space.

- Download and install OpenJDK™ 8 (Hot Spot) from <https://adoptopenjdk.net/>.
- Download and install Apache CXF software from <https://cxf.apache.org/download>. Choose the latest release of version 3.2.
- Make note of the installation folders created by these programs. Set the paths to the these variables, `jdk = 'YOUR_JDK_PATH'` and `cxf = 'YOUR_CXF_PATH'`, then call:

```
matlab.wsdL.setWSDLToolPath('JDK',jdk,'CXF',cxf)
```

See Also

`matlab.wsdL.setWSDLToolPath`

External Websites

- <https://adoptopenjdk.net/>
- <https://cxf.apache.org/download>

Display a World Map

This example shows how to access imagery from the United States Geological Survey (USGS) National Map SOAP server. To create the map, you need the following information.

- Get a map tile.
- Get the map name.
- Get the format of the tiles.

This example shows you how to call functions in the USGS web service, `USGSImageryOnly_MapServer`, to get this information.

Install the Java JDK and Apache CXF programs and set the tool paths to run this example.

```
p = matlab.wsdL.setWSDLToolPath;
if (isempty(p.JDK) || isempty(p.CXF))
    disp('Install the Java Development Kit (JDK) and Apache CXF programs.')
    disp('See the Set Up WSDL Tools link at the end of this example.')
else
    disp('Paths set to:')
    matlab.wsdL.setWSDLToolPath
end
```

Change your current folder to a writable folder.

Assign the WSDL URL.

```
wsdLFile = ...
'http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapServer?wsdl';
```

Create the class files for the client.

```
matlab.wsdL.createWSDLClient(wsdLFile)
```

```
Created USGSImageryOnly_MapServer.
.\USGSImageryOnly_MapServer.m
.\+wsdl
```

In order to use `USGSImageryOnly_MapServer`, you must run `javaaddpath('.\+wsdl\mapserver.jar')`.

```
ans =
```

```
@USGSImageryOnly_MapServer
```

Add the jar files to the Java path.

```
javaaddpath('.\+wsdl\mapserver.jar')
```

Start the service.

```
wsdl = USGSImageryOnly_MapServer;
```

Explore the service.

```
help USGSImageryOnly_MapServer
```

```
USGSImageryOnly_MapServer  A client to connect to the USGSImageryOnly_MapServer service
SERVICE = USGSImageryOnly_MapServer  connects to http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapS
```

```
To communicate with the service, call a function on the SERVICE:
```

```
[...] = FUNCTION(SERVICE,arg,...)
```

```
See doc USGSImageryOnly_MapServer for a list of functions.
```

Click the link `doc USGSImageryOnly_MapServer`. MATLAB opens a reference page for `USGSImageryOnly_MapServer` in the Help browser.

Read the documentation for the required inputs to the `GetMapTile` function.

```
help GetMapTile
```

```
--- help for USGSImageryOnly_MapServer/GetMapTile ---
```

```
GetMapTile
```

```
Result = GetMapTile(obj,MapName,Level,Row,Column,Format)
```

```
Inputs:
```

```
obj - USGSImageryOnly_MapServer object
```

```
MapName - string
```

```
Level - numeric scalar (XML int)
```

```
Row - numeric scalar (XML int)
```

```
Column - numeric scalar (XML int)
```

```
Format - string
```

```
Output:
```

```
Result - vector of numbers 0-255 (XML base64Binary)
```

```
See also USGSImageryOnly_MapServer.
```

You need `MapName`, `Level`, `Row`, `Column`, and `Format` input arguments.

Read the documentation for a function that provides a map name, `GetDefaultMapName`.

```
help GetDefaultMapName
```



```
--- help for USGSImageryOnly_MapServer/GetDefaultMapName ---
```

```
GetDefaultMapName
Result = GetDefaultMapName(obj)
Inputs:
  obj - USGSImageryOnly_MapServer object
Output:
  Result - string
```

See also USGSImageryOnly_MapServer.

This function provides a map name.

Read the documentation for a function that provides a map format information, `GetTileImageInfo`.

help `GetTileImageInfo`

```
--- help for USGSImageryOnly_MapServer/GetTileImageInfo ---
```

```
GetTileImageInfo
Result = GetTileImageInfo(obj,MapName)
Inputs:
  obj - USGSImageryOnly_MapServer object
  MapName - string
Output:
  Result - TileImageInfo object
```

See also USGSImageryOnly_MapServer.

This function returns a `TileImageInfo` object.

Read the documentation for the `TileImageInfo` object by clicking the link in the help display to `TileImageInfo`.

```
TileImageInfo(CacheTileFormat,CompressionQuality,Antialiasing) TileImageInfo object for use with USGSImageryOnly_MapServer
```

```
CacheTileFormat - string
  The cache tile format.
CompressionQuality - numeric scalar (XML int)
  The cache tile image compression quality.
Antialiasing - string
```

See also
USGSImageryOnly_MapServer.

MATLAB opens a document in the Help browser. The format information is `CacheTileFormat`.

Create the JPEG data. The following code requires knowledge of the JPEG image format and the tiling scheme used by the USGS server.

```
% Get the default map name.
defaultMapName = GetDefaultMapName(wsdL);

% Get the map count.
count = GetMapCount(wsdL);

% Get the map name. There is only one map (count value),
% but the index is zero-based.
mapName = GetMapName(wsdL, count-1);

% Get information about the tiles.
tileImageInfo = GetTileImageInfo(wsdL, mapName);

% Get the format of the data.
format = tileImageInfo.CacheTileFormat;

% Since format is specified as 'Mixed' it implies that
% the result of GetMapTile is a JPEG-encoded stream.
% The map tiles are organized with the lowest level as
% the lowest level of detail and the tiles use
% zero-based indexing.
level = 0;
row = 0;
col = 0;
jpeg = GetMapTile(wsdL, mapName, level, row, col, format);
```

Write the JPEG-encoded data to a file. Use `imread` to read and decode the JPEG data and return an M-by-N-by-3 `uint8` matrix.

```
ext = '.jpg';
tilename = ['USGSImageryOnly_MapServer' '0_0_0' ext];
fid = fopen(tilename, 'w');
fwrite(fid, jpeg)
fclose(fid)
```

View the map.

```
tileImage = imread(tilename);  
figure  
imshow(tileImage)
```

See Also

Related Examples

- “Set Up WSDL Tools” on page 19-34

Using WSDL Web Service with MATLAB

In this section...
“What Are Web Services in MATLAB?” on page 19-40
“What Are WSDL Documents?” on page 19-40
“What You Need to Use WSDL with MATLAB” on page 19-41

What Are Web Services in MATLAB?

Web services allow applications running on disparate computers, operating systems, and development environments to communicate with each other. There are two ways to use web services in MATLAB. When the service you want to use provides:

- RESTful (Representational state transfer), use the `webread` and `websave` functions in “Web Access”.
- Web Services Description Language (WSDL) document, use the MATLAB `matlab.wsdL.createWSDLCClient` function, described in the following topics.

What Are WSDL Documents?

Using a web service based on Web Services Description Language (WSDL) document technologies, client workstations access and execute APIs residing on a remote server. The client and server communicate via XML-formatted messages, following the W3C® SOAP protocol, and typically via the HTTP protocol.

Using the WSDL interface, MATLAB acts as a web service client, providing functions you use to access existing services on a server. The functions facilitate communication with the server, relieving you of the need to work with XML, complex SOAP messages, and special web service tools. Through these functions, you use services in your normal MATLAB environment, such as in the Command Window and in MATLAB programs you write.

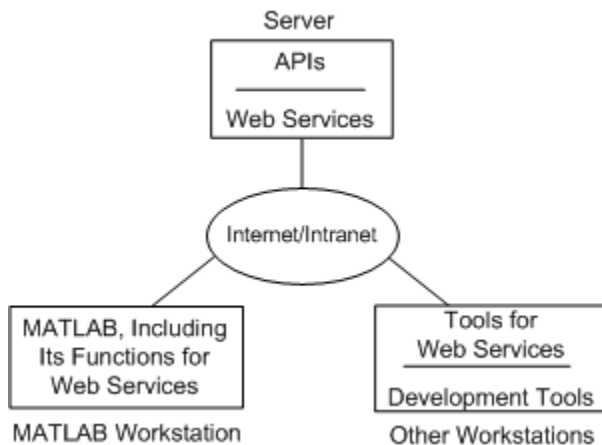


Diagram Showing Web Services in MATLAB

An organization that wants to make APIs available to disparate clients creates the APIs and related web service facilities for the server. Organizations either choose to make the services available only to local clients via the organization's intranet, or offer them to the general public via the web.

What You Need to Use WSDL with MATLAB

You need to find out from your own organization and the organizations you work with if they provide a web service of interest to you. There are publicly available services, some for free and some provided for a fee.

Functions for MATLAB WSDL web services work with services that comply with the Basic Profile 1 to SOAP Binding specification.

You need to know basic information about the service you want to use, provided by the documentation for the service.

You need access to the server from the workstation where you use MATLAB. If there is a proxy server, provide any required settings using web preferences. To do so, see “Specify Proxy Server Settings for Connecting to the Internet”.

To get started, see “WSDL (Web Services Description Language)”.

See Also

`matlab.wsdL.createWSDClient` | `webread` | `websave`

Access Services That Use WSDL Documents

A WSDL document uses a standard format to describe server operations, arguments, and transactions. The `matlab.wsdl.createWSDLClient` function creates a MATLAB class that allows you to use the server APIs.

To use the `matlab.wsdl.createWSDLClient` function, identify the location of the service WSDL document. The function works with WSDL documents that comply with the WS-I 1.0 standard and use one of these forms: RPC-literal, Document-literal, or Document-literal-wrapped. `matlab.wsdl.createWSDLClient` does not support RPC-encoded.

You must download supported versions of the OpenJDK™ or Oracle Java JDK and the Apache CXF programs.

To access a service:

- 1 Install and/or locate the Java JDK and Apache CXF programs.
- 2 Set the paths to the JDK and CXF programs using the `matlab.wsdl.setWSDLToolPath` function. Values for the paths are saved across sessions in your user preferences, so you only need to specify them once.
- 3 Change the MATLAB current folder to the location where you want to use the files generated from the WSDL document. You must have write-permission for this folder.
- 4 Run `matlab.wsdl.createWSDLClient`, supplying the WSDL document location, which can be a URL or a path to a file.

The function converts the server APIs to a MATLAB class, and creates a class folder in the current folder. The class folder contains methods for using the server APIs. The function always creates a constructor method that has the same name as the class.

You only run the `matlab.wsdl.createWSDLClient` function once. Then you can access the class anytime.

- 5 Create an object of the class whenever you want to use the operations of the service.
- 6 View information about the class to see what methods (operations) are available for you to use.
- 7 Use the methods of the object to run applications on and exchange data with the server.

MATLAB automatically converts XML data types to MATLAB types, and conversely.

See Also

`matlab.wsdL.createWSDLClient` | `matlab.wsdL.setWSDLToolPath`

Related Examples

- “Set Up WSDL Tools” on page 19-34

More About

- “XML-MATLAB Data Type Conversion” on page 19-47
- “Limitations to WSDL Document Support” on page 19-49

Error Handling

In this section...

- “Considerations Using Web Services” on page 19-45
- “Error Handling with try/catch Statements” on page 19-45
- “Use a Local Copy of the WSDL Document” on page 19-45
- “Java Errors Accessing Service” on page 19-46
- “Anonymous Types Not Supported” on page 19-46

Considerations Using Web Services

When creating MATLAB files that rely on a web service, consider the following:

- A server issues a time-out error. Repeat the MATLAB command.
- Internet performance might make your application performance unpredictable.
- Conventions and established procedures for services and related technologies, like WSDL and SOAP, are still evolving. You might find inconsistencies or unexpected behavior when using a web service.
- A service might change over time, which can impact its usage and results in MATLAB.
- A server issues other unrecoverable errors.

Error Handling with try/catch Statements

Use the `error` function in try/catch statements to catch errors that result from method calls or from the `matlab.wsdL.createWSDLCClient` function.

Use a Local Copy of the WSDL Document

You can achieve better performance if you create a local copy and use the local copy instead of the version at the URL.

```
wsdLURL = ...  
['http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapServer?wsdl'];  
wsdlFile = 'USGSImageryOnly_MapServer';  
if ~exist(wsdLFile, 'file')
```

```
    websave(wsdFile, wsdURL)
end
```

Use this strategy when you do not need immediate access to data at the URL.

Java Errors Accessing Service

Once you access a service from MATLAB using the generated client code, you might get Java errors if:

- The WSDL for the service changes and you run `matlabl.wsd.createWSDLClient` again for the same service in the same MATLAB session.
- You try to access the service using the regenerated code.

These errors are likely to occur if you modify the service between successive calls to `matlabl.wsd.createWSDLClient`.

If you change a service you already accessed or generate the class files in another location, restart MATLAB.

Anonymous Types Not Supported

Anonymous XML types are unnamed types embedded in other types.

For more information, see the IBM® article at “Web services hints and tips: avoid anonymous types”.

See Also

error

More About

- “Exception Handling in a MATLAB Application”

External Websites

- W3C status codes for HTTP errors

XML-MATLAB Data Type Conversion

MATLAB SOAP functions automatically convert XML data types used in SOAP messages to and from MATLAB types (classes). The following table contains the XML type and the corresponding MATLAB type for scalar values used in a WSDL document.

XML Schema Type	MATLAB Type Returned—Scalar
boolean	logical
byte	int8
unsignedByte	uint8
short	int16
unsignedShort	uint16
int	int32
unsignedInt	uint32
long	int64
unsignedLong	uint64
float	double
double	double
string	char vector
gYear, gMonth, gDay, gYearMonth, gMonthDay	calendarDuration array
dateTime	dateTime
date	dateTime with Year, Month, Day fields undefined.
time	dateTime with Hours, Minutes, Seconds fields undefined.
duration	duration if no year, month or day calendarDuration otherwise
NOTATION, QName	Character vector containing a legal QName
hexbinary, base64Binary	N-by-1 vector of uint8 representing byte values (0-255) of encoded data

XML Schema Type	MATLAB Type Returned—Scalar
decimal, integer, nonPositiveInteger, nonNegativeInteger, positiveInteger, negativeInteger	double array

The following table contains the XML type and the corresponding MATLAB type for arrays.

XML Schema Type—Array	MATLAB Type Returned—Array
array of string	N-by-1 cell array of characters
array of any <i>type</i>	N-by-1 vector of specified <i>type</i>
array of hexbinary, base64Binary	Not supported

Limitations to WSDL Document Support

In this section...

“Unsupported WSDL Documents” on page 19-49

“Documents Must Conform to Wrapper Style” on page 19-51

“SOAP Header Fields Not Supported” on page 19-52

Unsupported WSDL Documents

- RPC-encoded WSDL documents. For these documents, use `createClassFromWsdL`.
- Documents that the Apache CXF program cannot compile into complete code.
- Documents that import other WSDL documents that contain WSDL type definitions.

Workaround

Move all schema definitions from the imported files into the top-level WSDL file. Schema definitions appear as `schema` elements inside `types` elements.

If the imported files are not local, copy them locally and modify the `import` elements of all the files to point to the local files.

For example, consider the following top-level file.

```
<definitions>
  <import location="http://foo/bar?wsdl" />
  <types>
    ...top level type definitions...
  </types>
</definitions>
```

Download the file in the `location` attribute, `http://foo/bar?wsdl`, to a local file, and save it as `imported_file`. This file contains information similar to the following.

```
<wsdl:types>
  ...low level type definitions...
  <xsd:schema>
    ...schema definitions...
  </xsd:schema>
</wsdl:types>
```

Look for `types` and `schema` elements. The text prefixes, `wSDL` and `xsd`, do not have standard naming conventions and might not appear at all. Do not download `import` elements *within* the schema definitions. Move all `schema` elements in the imported file, including the opening and closing tags, from the imported file to the end of the `types` element of the top-level file. Then delete the elements from the imported file. Do not modify existing `schema` elements in the top-level file. If the top-level file already contains a `types` section, add the `schema` elements to the existing `types` content. Otherwise, create a `types` section within the `definitions` element of the top-level file to contain the `schema` elements.

The top-level file now contains the following.

```
<definitions>
  <import location="imported_file" />
  <types>
    ...top level type definitions...
    <xsd:schema>
      ...schema definitions...
    </xsd:schema>
  </types>
</definitions>
```

The `imported_file` file contains the following.

```
<wSDL:types>
  ...low level type definitions...
</wSDL:types>
```

There must be exactly one `types` element in the top-level file inside the `definitions` element, containing all the `schema` defined in the imported WSDL files. None of the imported WSDL files should contain any `schema` elements.

- On Windows, documents that import other WSDL documents might fail if the imported URI contains certain punctuation characters.
- Some documents with messages containing multiple parts.
- Some documents with schemas containing anonymous complex types.
- Some documents defining an input parameter to an operation as a simple type. When you invoke such an operation, for example `GetMyOp`, MATLAB displays one of the following errors.

```
Error using xxx/GetMyOp. Too many input arguments.
```

Or:

```
Error using matlab.internal.callJava
No GetMyOp method with appropriate signature exists in Java class $Proxy57
```

- If the WSDL document defines schema types in multiple namespaces, an error might occur if types in different namespaces have the same names. Multiple namespaces usually occur when using `import` statements. MATLAB displays messages similar to the following.

```
Error using matlab.wsdL.createWsdLClient (line 239)
Error processing WSDL:
file:/l:/02090000/incoming/service_w0_x0.xsd [149,4]: Two declarations cause a collision in the ObjectFactory class.
```

To work around this problem, copy the imported files and edit them to rename the conflicting types.

- XML Schema element `all` not recognized.

Workaround

If a `complexType` is defined using `all`, then none of the child elements appear in the generated MATLAB class for the type. In this example, the `Id` and `Name` elements do not appear as a properties of the `Record` class.

```
<xsd:complexType name="Record">
  <xsd:all>
    <xsd:element name="Id" type="xsd:string"/>
    <xsd:element name="Name" type="xsd:string"/>
  </xsd:all>
</xsd:complexType>
```

To work around, copy the WSDL file locally and replace `xsd:all` with `xsd:sequence`.

```
<xsd:complexType name="Record">
  <xsd:sequence>
    <xsd:element name="Id" type="xsd:string"/>
    <xsd:element name="Name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Documents Must Conform to Wrapper Style

Operations defined in the WSDL must conform to the rules for wrapper style, as described by item (ii) in section 2.3.1.2 of The Java API for XML Web Services (JAX-WS) 2.0. Error messages similar to the following are indications of this problem.

```
Error using matlab.internal.callJava
No authenticate method with appropriate signature exists in Java class com.sun.proxy.$Proxy55

Error in Service/Authenticate (line 107)
    matlab.internal.callJava('authenticate',obj.PortObj,fromMATLAB({'Authenticate','user'},
    user,'string',false,false),...
```

To work around this issue, edit the WSDL to conform to the wrapper style rules, or edit the generated MATLAB code to instantiate and pass in the Java class object that contains the parameters.

SOAP Header Fields Not Supported

It is not possible to send messages that require Simple Object Access Protocol (SOAP) header elements. Only SOAP body elements are supported.

Manually Redirect HTTP Messages

You can use cookies to manually handle redirects. This example uses websites named `http://www.somewebsite1.com` and `http://www.somewebsite2.com`.

```
import matlab.net.http.*
import matlab.net.http.field.*
r = RequestMessage;
[resp,~,history] = r.send('http://www.somewebsite1.com');
cookieInfos = CookieInfo.collectFromLog(history);
if ~isempty(cookieInfos)
    cookies = [cookieInfos.Cookie];
end
r2 = RequestMessage([],CookieField(cookies));
% This is the new location for redirection
[resp,~,history] = r.send('http://www.somewebsite2.com');
```

See Also

Python Interface Topics

- “Install Supported Python Implementation” on page 20-3
- “Call Python from MATLAB” on page 20-4
- “Call User-Defined Python Module” on page 20-7
- “Use Python Numeric Types in MATLAB” on page 20-9
- “Call Python Methods with Numeric Arguments” on page 20-10
- “Default Numeric Types” on page 20-11
- “Use Python array Types in MATLAB” on page 20-12
- “Pass MATLAB Character Vector to Python Method” on page 20-13
- “Use Python str Type in MATLAB” on page 20-14
- “Pass MATLAB Backslash Control Character” on page 20-16
- “Create Python list Variable” on page 20-17
- “Use Python list Type in MATLAB” on page 20-18
- “Use Python List of Numeric Types in MATLAB” on page 20-20
- “Pass Cell Array as Python Sequence Type” on page 20-22
- “Read Element of Nested list Type” on page 20-24
- “Create Python tuple Variable” on page 20-25
- “Create Singleton Python tuple Variable” on page 20-26
- “Create Python dict Variable” on page 20-27
- “Pass dict Argument to Python Method” on page 20-28
- “Use Python dict Type in MATLAB” on page 20-29
- “Convert Python dict Type to MATLAB Structure” on page 20-31
- “Pass Keyword Arguments” on page 20-32
- “Pass Python Function to Python map Function” on page 20-33
- “Index into Python String” on page 20-34
- “Index into Python List” on page 20-35
- “Index into Python Tuple” on page 20-37

- “Index into Python dict” on page 20-38
- “Use Python List as Values in for Loop” on page 20-40
- “Display Stepped Range of Python Elements” on page 20-41
- “Access Elements in Python Container Types” on page 20-42
- “View Python Numeric Values” on page 20-45
- “Call Methods on Python Variables” on page 20-47
- “Reload Modified User-Defined Python Module” on page 20-49
- “System and Configuration Requirements” on page 20-51
- “Create a Python Object” on page 20-53
- “Pass Data to Python” on page 20-56
- “Troubleshooting Multidimensional Array Argument Errors” on page 20-61
- “Unsupported MATLAB Types” on page 20-62
- “Handle Data Returned from Python” on page 20-63
- “How MATLAB Represents Python Operators” on page 20-66
- “Execute Callable Python Object” on page 20-68
- “Understanding Python and MATLAB import Commands” on page 20-69
- “List, Tuple, and Dictionary Types” on page 20-71
- “Limitations to Python Support” on page 20-72
- “Limitations to Indexing into Python Objects” on page 20-74
- “Undefined variable “py” or function “py.command”” on page 20-75
- “Help for Python Functions” on page 20-78
- “Handle Python Exceptions” on page 20-80
- “Troubleshooting Error Messages” on page 20-81
- “Using Python Data in MATLAB” on page 20-83
- “Call Python eval Function” on page 20-84
- “Precedence Order of Methods and Functions” on page 20-86
- “Python Function Arguments” on page 20-87
- “Out-of-Process Execution of Python Functionality” on page 20-90

Install Supported Python Implementation

- Access <https://www.python.org/downloads/> and scroll down to the **Looking for a specific release** section.
- Find the version you want and click **Download**. For a list of the currently supported versions of Python, see “Call Python Library Functions”.
- Click the format you want for the 64-bit version, and follow the online instructions.

See Also

pyenv

More About

- “MATLAB Cannot Find Python” on page 20-75
- “System and Configuration Requirements” on page 20-51

Call Python from MATLAB

This example shows how to use Python® language functions and modules within MATLAB®. The example calls a text-formatting module from the Python standard library.

MATLAB supports the reference implementation of Python, often called CPython, versions 2.7, 3.6, and 3.7. If you are on a Mac or Linux platform, you already have Python installed. If you are on Windows, you need to install a distribution, such as those found at <https://www.python.org/download>, if you have not already done so. For more information, see “Install Supported Python Implementation” on page 20-3.

Call a Python Function to Wrap Text in a Paragraph

MATLAB has equivalencies for much of the Python standard library, but not everything. For example, `textwrap` is a module for formatting blocks of text with carriage returns and other conveniences. MATLAB also provides a `textwrap` function, but it only wraps text to fit inside a UI control.

Create a paragraph of text to play with.

```
T = 'MATLAB(R) is a high-level language and interactive environment for numerical comp
```

Convert a Python String to a MATLAB String

Call the `textwrap.wrap` function by typing the characters `py.` in front of the function name. Do not type `import textwrap`.

```
wrapped = py.textwrap.wrap(T);  
whos wrapped
```

Name	Size	Bytes	Class	Attributes
wrapped	1x7	8	py.list	

`wrapped` is a Python list, which is a list of Python strings. MATLAB shows this type as `py.list`.

Convert `py.list` to a cell array of Python strings.

```
wrapped = cell(wrapped);  
whos wrapped
```

Name	Size	Bytes	Class	Attributes
wrapped	1x7	840	cell	

Although `wrapped` is a MATLAB cell array, each cell element is a Python string.

```
wrapped{1}
```

```
ans =
```

```
Python str with no properties.
```

```
    MATLAB(R) is a high-level language and interactive environment for
```

Convert the Python strings to MATLAB strings using the `char` function.

```
wrapped = cellfun(@char, wrapped, 'UniformOutput', false);
wrapped{1}
```

```
ans =
```

```
    'MATLAB(R) is a high-level language and interactive environment for'
```

Now each cell element is a MATLAB string.

Customize the Paragraph

Customize the output of the paragraph using keyword arguments.

The previous code uses the `wrap` convenience function, but the module provides many more options using the `py.textwrap.TextWrapper` functionality. To use the options, call `py.textwrap.TextWrapper` with keyword arguments described at <https://docs.python.org/2/library/textwrap.html#textwrap.TextWrapper>.

Create keyword arguments using the MATLAB `pyargs` function with a comma-separated list of name/value pairs. `width` formats the text to be 30 characters wide. The `initial_indent` and `subsequent_indent` keywords begin each line with the comment character, `%`, used by MATLAB.

```
tw = py.textwrap.TextWrapper(pyargs(...
    'initial_indent', '% ', ...
```

```
    'subsequent_indent', '% ', ...  
    'width', int32(30));  
wrapped = wrap(tw,T);
```

Convert to a MATLAB argument and display the results.

```
wrapped = cellfun(@char, cell(wrapped), 'UniformOutput', false);  
fprintf('%s\n', wrapped{:})
```

```
% MATLAB(R) is a high-level  
% language and interactive  
% environment for numerical  
% computation, visualization,  
% and programming. Using  
% MATLAB, you can analyze  
% data, develop algorithms,  
% and create models and  
% applications. The language,  
% tools, and built-in math  
% functions enable you to  
% explore multiple approaches  
% and reach a solution faster  
% than with spreadsheets or  
% traditional programming  
% languages, such as C/C++ or  
% Java(TM).
```

Learn More

It is sufficient to remember that Python is yet another potential source of libraries for the MATLAB user. If you want to learn about moving data between MATLAB and Python, including Python data types such as tuples and dictionaries, see “Python Libraries”.

Call User-Defined Python Module

This example shows how to call methods from the following Python module. This module is used by examples in the documentation.

This example explains how to create the module in MATLAB. If you create `mymod.py` in a Python editor, be sure that the module is on the Python search path. This example also explains how to get help for calling the function, if you are not an experienced Python user.

Change your current folder to a writable folder.

Open a new file in MATLAB Editor.

Copy these commands and save the file as `mymod.py`.

```
# mymod.py
"""Python module demonstrates passing MATLAB types to Python functions"""
def search(words):
    """Return list of words containing 'son'"""
    newlist = [w for w in words if 'son' in w]
    return newlist

def theend(words):
    """Append 'The End' to list of words"""
    words.append('The End')
    return words
```

From the MATLAB command prompt, add the current folder to the Python search path.

```
if count(py.sys.path, '') == 0
    insert(py.sys.path, int32(0), '');
end
```

To learn how to call the function, read the function signature for the `search` function in the `mymod.py` source file. The function takes one input argument, `words`.

```
def search(words):
```

Read the function help in the `mymod.py` source file. According to the Python website documentation, help is in “a string literal that occurs as the first statement in a module, function, class, or method definition.” The help for `search` is:

```
"""Return list of words containing 'son'"""
```

The function returns a list.

Create an input argument, a list of names, in MATLAB.

```
N = py.list({'Jones', 'Johnson', 'James'})
```

```
N =
```

```
Python list with no properties.
```

```
['Jones', 'Johnson', 'James']
```

Call the `search` function. Type `py.` in front of the module name and function name.

```
names = py.mymod.search(N)
```

```
names =
```

```
Python list with no properties.
```

```
['Johnson']
```

The function returns a `py.list` value.

The original input `N` is unchanged.

```
N
```

```
N =
```

```
Python list with no properties.
```

```
['Jones', 'Johnson', 'James']
```

See Also

Related Examples

- “System and Configuration Requirements” on page 20-51
- “Reload Modified User-Defined Python Module” on page 20-49

Use Python Numeric Types in MATLAB

This example shows how to use Python® numbers in MATLAB®. The trigonometry functions in the `math` module return Python `float` types.

```
pynum = py.math.radians(90)
```

```
pynum = 1.5708
```

MATLAB automatically converts this type to `double`.

```
class(pynum)
```

```
ans =  
'double'
```

See Also

More About

- “Handle Data Returned from Python” on page 20-63

Call Python Methods with Numeric Arguments

This example shows how to call the Python `math.fsum` function, which sums the floating-point values in an iterable input argument.

Open the MATLAB® `patients.mat` data file and read the numeric array `Height`.

```
load patients.mat
class(Height)
```

```
ans =
'double'
```

MATLAB automatically converts the numeric values to Python numeric values. However, `Height` is a 100-by-1 array, and MATLAB must pass a 1-by-N array to a Python iterable argument.

```
size(Height)
```

```
ans = 1×2
    100     1
```

Transform `Height` to a 1-by-N matrix before calling `fsum`.

```
py.math.fsum(Height')
```

```
ans = 6707
```

See Also

More About

- “Default Numeric Types” on page 20-11
- “Pass Data to Python” on page 20-56

Default Numeric Types

By default, a number in MATLAB is a `double` type. By default, a number (without a fractional part) in Python is an integer type. This difference can cause confusion when passing numbers to Python functions.

For example, when you pass the following MATLAB numbers to the Python `datetime` function, Python reads them as `float` types.

```
d = py.datetime.date(2014,12,31)
```

```
Python Error: TypeError: integer argument expected, got float
```

Explicitly convert each number to an integer type:

```
d = py.datetime.date(int32(2014),int32(12),int32(31))
```

```
d =
```

```
Python date with properties:
```

```
    day: 31
  month: 12
   year: 2014
```

```
2014-12-31
```

Use Python array Types in MATLAB

This example shows how to sum the elements of a Python array `.array` of type `double`.

Suppose that you have a Python function that returns the following array, `P`.

```
P =
```

```
Python array with properties:
```

```
    itemsize: 8  
    typecode: [1x1 py.str]  
  
array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
```

Convert `P` to a MATLAB array of type `double`.

```
A = double(P);
```

Sum the elements of `A`.

```
sum(A)
```

```
ans =
```

```
    15
```

See Also

More About

- “Unsupported MATLAB Types” on page 20-62

Pass MATLAB Character Vector to Python Method

This example shows how to display the contents of a folder using the Python `os.listdir` method. The `listdir` method returns a list containing the names of the entries in the folder given by the path input argument.

Create a MATLAB variable representing a valid folder.

```
folder = fullfile(matlabroot, 'help', 'examples');
```

Pass `folder` to the `os.listdir` function. MATLAB automatically converts `folder` to the Python `str` type.

```
F = py.os.listdir(folder)
```

```
F =
```

```
Python list with no properties.
```

```
['graphics', 'graphics2', 'matlab']
```

MATLAB displays a list of folders, based on your product.

See Also

More About

- “Pass Data to Python” on page 20-56

Use Python str Type in MATLAB

This example shows how to use the Python® path separator character (;). In MATLAB®, a Python character is a `py.str` variable.

```
p = py.os.path.pathsep  
  
p =  
    Python str with no properties.  
  
    ;
```

MATLAB uses the same path separator character, ;.

```
c = pathsep  
  
c =  
    ';'.
```

Compare the MATLAB variable type to the Python type. A `py.str` type is not equivalent to a MATLAB `char` type.

```
isequal(class(p),class(c))  
  
ans = logical  
     0
```

Convert `p` to a MATLAB type and append the character to a file name.

```
f = ['myfile' char(p)]  
  
f =  
    'myfile;'
```

See Also

More About

- “Explicit Type Conversions” on page 20-63

- “Index into Python String” on page 20-34

Pass MATLAB Backslash Control Character

This example shows how to pass the backslash control character (\) as a Python® `str` type.

Insert the new line control character `\n` by calling the MATLAB® `sprintf` function. Python replaces `\n` with a new line.

```
py.str(sprintf('The rain\nin Spain.'))
```

```
ans =  
Python str with no properties.  
  
The rain  
in Spain.
```

Without the `sprintf` function, both MATLAB and Python interpret `\` as a literal backslash.

```
py.str('The rain\nin Spain.')
```

```
ans =  
Python str with no properties.  
  
The rain\nin Spain.
```

Pass this string to a Python string method `split`. Python treats the MATLAB character vector as a *raw string* and adds a `\` character to preserve the original backslash.

```
split(py.str('The rain\nin Spain.'))
```

```
ans =  
Python list with no properties.  
  
['The', 'rain\\n', 'Spain.']
```

Create Python list Variable

This example shows how to create a `list` variable to pass to a Python® function.

```
students = py.list({'Robert', 'Mary', 'Joseph'})  
  
students =  
    Python list with no properties.  
  
    ['Robert', 'Mary', 'Joseph']
```

Display the number of students in the list.

```
n = py.len(students)  
  
n = int64  
    3
```

See Also

More About

- “List, Tuple, and Dictionary Types” on page 20-71
- “Pass Data to Python” on page 20-56

Use Python list Type in MATLAB

This example shows how to display folders on your system using the MATLAB `disp` function on the Python `sys.path` variable. `sys.path` is a `list` type.

Create a Python list, `P`, of folders on the Python search path.

```
P = py.sys.path;  
class(P)
```

```
ans =
```

```
py.list
```

Convert list `P` to a MATLAB type. The first step is to convert the list to a cell array of folder names, `cP`.

```
cP = cell(P);  
class(cP)
```

```
ans =
```

```
cell
```

Each folder name is a Python string.

```
class(cP{1})
```

```
ans =
```

```
py.str
```

Convert the Python strings in `cP` to MATLAB data using the `char` function. Put the values into a new cell array, `cellP`. Display the folder names.

```
cellP = cell(1, numel(cP));  
for n=1:numel(cP)  
    strP = char(cP{n});  
    cellP(n) = {strP};  
    disp(strP)  
end
```

```
C:\Python27\lib\site-packages\protobuf-2.5.0-py2.7.egg  
C:\windows\system32\python27.zip  
C:\Python27\DLLs
```

```
C:\Python27\lib  
C:\Python27\lib\plat-win  
C:\Python27\lib\lib-tk  
C:\Python27  
C:\Python27\lib\site-packages
```

MATLAB displays information specific to your Python installation.

Alternatively, create `cellP` using the `cellfun` function to combine the conversion functions.

```
cellP = cellfun(@char,cell(P), 'UniformOutput', false);
```

Display the folder names.

```
for n=1:py.len(P)  
    disp(cellP{n})  
end
```

See Also

[cell](#) | [cellfun](#)

More About

- “Index into Python List” on page 20-35
- “Handle Data Returned from Python” on page 20-63

Use Python List of Numeric Types in MATLAB

This example shows how to convert a Python® list of numeric values into a MATLAB® array of double.

A Python list contains elements of any type and can contain elements of mixed types. The MATLAB `double` function used in this example assumes that all elements of the Python list are numeric.

Suppose that you have a Python function that returns the following list of integers P. To run this example, create the variable.

```
P = py.list({int32(1), int32(2), int32(3), int32(4)})  
P =  
    Python list with no properties.  
    [1, 2, 3, 4]
```

Display the numeric type of the values.

```
class(P{1})
```

```
ans =  
'int64'
```

Convert P to a MATLAB cell array.

```
cP = cell(P);
```

Convert the cell array to a MATLAB array of double.

```
A = cellfun(@double,cP)
```

```
A = 1×4
```

```
    1    2    3    4
```

See Also

`cell` | `cellfun`

Related Examples

- “Use Python array Types in MATLAB” on page 20-12

Pass Cell Array as Python Sequence Type

This example shows how to pass a MATLAB cell array to a Python function.

The following Python module contains the function `theend`, which appends text to the end of a list. Create a text file, copy this Python module, and save the file as `mymod.py` in a writable folder, for example, your `prefdir` folder.

```
# mymod.py
"""Python module demonstrates passing MATLAB types to Python functions"""
def search(words):
    """Return list of words containing 'son'"""
    newlist = [w for w in words if 'son' in w]
    return newlist

def theend(words):
    """Append 'The End' to list of words"""
    words.append('The End')
    return words
```

Add the folder containing `mymod.py` to the Python search path using the `append` method of the `list` type.

```
P = py.sys.path;
append(P,prefdir);
```

Open the MATLAB sample data file, creating the cell array, `LastName`.

```
load patients.mat
```

Convert the cell array to a 1-by-N `py.list` array.

```
L = py.list(LastName');
class(L)
```

```
ans =
```

```
py.list
```

Call `mymod.theend` on the list.

```
py.mymod.theend(L);
```

The Python function does not return variable `L` as an output argument. Instead, the function updates the value by reference.

View the last item in the list.

```
L{end}
```

```
ans =
```

```
Python str with no properties.  
The End
```

You can pass variable L to other Python functions.

See Also

Related Examples

- “Use Python list Type in MATLAB” on page 20-18
- “Pass Data to Python” on page 20-56

Read Element of Nested list Type

This example shows how to access an element of a Python® list variable containing list elements.

```
matrix = py.list({{1, 2, 3, 4},{'hello','world'},{9, 10}});
```

Display element 'world' which is at index (2,2).

```
disp(char(matrix{2}{2}))
```

```
world
```

See Also

More About

- “Multilevel Indexing to Access Parts of Cells”

Create Python tuple Variable

This example shows how to create a tuple variable to pass to a Python® function.

```
student = py.tuple({'Robert',19,'Biology'})
```

```
student =  
    Python tuple with no properties.  
  
    ('Robert', 19.0, 'Biology')
```

See Also

More About

- “List, Tuple, and Dictionary Types” on page 20-71
- “Pass Data to Python” on page 20-56

Create Singleton Python tuple Variable

This example shows how to create a Python® tuple variable with a single element. A tuple with one element has a trailing comma.

```
subject = py.tuple({'Biology'})

subject =
    Python tuple with no properties.

    ('Biology',)
```

Create Python dict Variable

This example shows how to create a dict variable to pass to a Python® function.

```
studentID = py.dict(pyargs('Robert',357,'Mary',229,'Jack',391))
```

```
studentID =
```

```
    Python dict with no properties.
```

```
    {'Robert': 357.0, 'Mary': 229.0, 'Jack': 391.0}
```

See Also

More About

- “List, Tuple, and Dictionary Types” on page 20-71
- “Pass Data to Python” on page 20-56

Pass dict Argument to Python Method

This example shows how to change a value in a dict variable using the Python® update method.

Create a menu of items and prices.

```
menu = py.dict(pyargs('soup',3.57,'bread',2.29,'bacon',3.91,'salad',5.00));
```

Update the price for bread using the Python dict type update method.

```
update(menu,py.dict(pyargs('bread',2.50)))
```

```
menu
```

```
menu =
```

```
    Python dict with no properties.
```

```
    {'soup': 3.57, 'bacon': 3.91, 'salad': 5.0, 'bread': 2.5}
```

See Also

More About

- “Pass Data to Python” on page 20-56

Use Python dict Type in MATLAB

This example shows how to convert numerical values of a Python `dict` variable into a MATLAB array.

Suppose that you have a Python function that returns menu items and prices in a dictionary `dict` type. To run this example in MATLAB using Python 2.7, create the variable `order`. When you display elements in a `dict` object, the order of the elements is variable.

```
order = py.dict(pyargs('soup',3.57,'bread',2.29,'bacon',3.91,'salad',5.00))
order =
```

```
Python dict with no properties.
```

```
{'soup': 3.57, 'bacon': 3.91, 'salad': 5.0, 'bread': 2.29}
```

A dictionary has pairs of keys and values. Display the menu items using the Python `keys` function.

```
keys(order)
```

```
ans =
Python list with no properties.
```

```
['soup', 'bacon', 'salad', 'bread']
```

Display all prices using the Python `values` function.

```
values(order)
```

```
ans =
Python list with no properties.
```

```
[3.57, 3.91, 5.0, 2.29]
```

Display the price of bacon. The type of variable `price` is double, which you can use in MATLAB.

```
price = order{'bacon'}
```

```
price =
3.9100
```

Convert order to a MATLAB variable.

```
myOrder = struct(order)
```

```
myOrder =  
    soup: 3.5700  
    bacon: 3.9100  
    salad: 5  
    bread: 2.2900
```

Display the price of bacon.

```
price = myOrder.bacon
```

```
price = 3.9100
```

See Also

struct

More About

- “Convert Python dict Type to MATLAB Structure” on page 20-31
- “Index into Python dict” on page 20-38
- “Use Python list Type in MATLAB” on page 20-18
- “Handle Data Returned from Python” on page 20-63

Convert Python dict Type to MATLAB Structure

This example shows how to plot numeric data from a Python dictionary.

Suppose a Python function returns a variable, `patient`, with the following values.

```
patient
```

```
patient =
```

```
Python dict with no properties.
```

```
{'test1': array('d', [79.0, 75.0, 73.0]),  
'test3': array('d', [220.0, 210.0, 205.0]),  
'test2': array('d', [180.0, 178.0, 177.5]),  
'name': 'John Doe'}
```

Convert `patient` to a MATLAB structure.

```
P = struct(patient)
```

```
P =
```

```
test1: [1x1 py.array.array]  
test3: [1x1 py.array.array]  
test2: [1x1 py.array.array]  
name: [1x8 py.str]
```

The values of the fields remain as Python types.

Plot the test results after converting the numeric data to type double.

```
bar([double(P.test1);double(P.test2);double(P.test3)])
```

MATLAB displays a bar graph.

See Also

More About

- “Handle Data Returned from Python” on page 20-63

Pass Keyword Arguments

The Python built-in `print` function has keyword arguments, `sep`, `end`, and `file`.

```
print(*objects, sep=' ', end='\n', file=sys.stdout)
```

The following examples use the default value for `file`.

Create some text variables.

```
x1 = py.str('c:');  
x2 = py.os.curdir;  
x3 = py.os.getenv('foo');  
py.print(x1,x2,x3)
```

```
c: . None
```

To display the values on separate lines, use newline, `\n`, as a separator.

```
py.print(x1,x2,x3,pyargs('sep',sprintf('\n')))
```

```
c:  
. None
```

Use the following statement to change `sep` to an empty string and change the `end` value to display `THE END`.

```
py.print(x1,x2,x3,pyargs('end', sprintf(' THE END\n'),'sep',py.str))
```

```
c:.None THE END
```

See Also

`pyargs`

More About

- “Pass Data to Python” on page 20-56

External Websites

- python.org `print` function

Pass Python Function to Python map Function

This example shows how to display the length of each word in a list.

Create a list of days of the work week.

```
days = py.list({'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'});
```

Display the length of each word by applying the Python `len` function to the `py.map` function. To indicate that `py.len` is a function, use the MATLAB function handle notation `@`.

```
py.map(@py.len, days)
```

```
ans =
```

```
Python list with no properties.
```

```
[6, 7, 9, 8, 6]
```

Python version 2.7 returns a list.

Python versions 3.x return a map object. To display the contents, type:

```
py.list(py.map(@py.len, days))
```

```
ans =
```

```
Python list with no properties.
```

```
[6, 7, 9, 8, 6]
```

See Also

External Websites

- python.org map function

Index into Python String

This example shows how to display the first character of a Python® `str` variable. The example compares indexing into a MATLAB® character vector with indexing into the Python variable.

Create a MATLAB character vector and display the first character.

```
str = 'myfile';  
str(1)
```

```
ans =  
'm'
```

Convert the character vector to a Python `str` type and display the first character.

```
pstr = py.str(str);  
pstr(1)
```

```
ans =  
  Python str with no properties.  
  
  m
```

See Also

More About

- “Use Python `str` Type in MATLAB” on page 20-14
- “Handle Data Returned from Python” on page 20-63

Index into Python List

This example shows how to display the last element in a Python® `list` variable. The example compares indexing into a MATLAB® cell array `C` with indexing into a Python list `li`.

Create a MATLAB cell array and display the last element. MATLAB returns a cell array.

```
C = {1,2,3,4};  
n = C(end)  
  
n = 1×1 cell array  
    {[4]}
```

Display the contents of the last element in the cell array.

```
n = C{end}  
  
n = 4
```

Convert the cell array to a Python list.

```
li = py.list(C)  
  
li =  
    Python list with no properties.  
  
    [1.0, 2.0, 3.0, 4.0]
```

Display the last element in the list. MATLAB returns a list.

```
n = li(end)  
  
n =  
    Python list with no properties.  
  
    [4.0]
```

Display the contents of the last element in the list. MATLAB converts the value to double.

```
n = li{end}  
  
n = 4
```

See Also

More About

- “Use Python List as Values in for Loop” on page 20-40
- “Access Elements in Python Container Types” on page 20-42
- “Handle Data Returned from Python” on page 20-63

Index into Python Tuple

This example shows how to display elements in a tuple.

Create a Python® tuple and display the first two elements. MATLAB® returns a tuple.

```
t = py.tuple({'a', 'bc', 1, 2, 'def'});
t(1:2)
```

```
ans =
    Python tuple with no properties.

    ('a', 'bc')
```

Convert to a MATLAB variable.

```
mt = cell(t)

mt = 1x5 cell array
    {1x1 py.str}    {1x2 py.str}    {[1]}    {[2]}    {1x3 py.str}
```

Display the second element.

```
mt{2}

ans =
    Python str with no properties.

    bc
```

See Also

More About

- “List, Tuple, and Dictionary Types” on page 20-71
- “Handle Data Returned from Python” on page 20-63

Index into Python dict

This example shows how to get and set values in a dictionary.

Create a dictionary variable in MATLAB®.

```
customers = py.dict

customers =
  Python dict with no properties.

  {}
```

Populate the dict variable with customer names and account numbers and display the results. The output depends on your Python® version.

```
customers{'Smith'} = int32(2112);
customers{'Anderson'} = int32(3010);
customers{'Audrey'} = int32(4444);
customers{'Megan'} = int32(5000);
customers

customers =
  Python dict with no properties.

  {'Audrey': 4444, 'Anderson': 3010, 'Smith': 2112, 'Megan': 5000}
```

Read the account number for customer Anderson.

```
acct = customers{'Anderson'}

acct = int64
  3010
```

Convert to a MATLAB variable.

```
C = struct(customers)

C = struct with fields:
  Audrey: 4444
Anderson: 3010
  Smith: 2112
```



```
Megan: 5000
```

```
acct = C.Anderson
```

```
acct = int64  
3010
```

See Also

More About

- “Use Python dict Type in MATLAB” on page 20-29
- “Handle Data Returned from Python” on page 20-63

Use Python List as Values in for Loop

This example shows how to display elements of a Python® list.

```
li = py.list({1,2,3,4});  
for n = li  
    disp(n{1})  
end  
  
1  
  
2  
  
3  
  
4
```

See Also

More About

- “Index into Python List” on page 20-35
- “Access Elements in Python Container Types” on page 20-42

Display Stepped Range of Python Elements

This example shows how to use an incremental step in indexing.

If you use slicing to access elements of a Python® object, the format in Python is `start:stop:step`. In MATLAB®, the syntax is of the form `start:step:stop`.

```
li = py.list({'a', 'bc', 1, 2, 'def'});  
li(1:2:end)
```

```
ans =  
    Python list with no properties.
```

```
    ['a', 1.0, 'def']
```

Access Elements in Python Container Types

In this section...

“Sequence Types” on page 20-42

“Mapping Types” on page 20-43

“Size and Dimensions” on page 20-43

“Array Support” on page 20-44

“Use Zero-Based Indexing for Python Functions” on page 20-44

To work with a Python variable in MATLAB, convert the Python object to a MATLAB array, and then index into the array as needed. You can also preserve the Python object without converting, for example, to pass the object to a Python method.

A Python container is typically a sequence type (list or tuple) or a mapping type (dict). In Python, use square brackets `[]` or the operator `.getitem` function to access an element in the container. Scalar string arguments can be used to index into the container.

Sequence Types

Python sequence types behave like MATLAB cell arrays.

Get a subsequence using smooth-parenthesis `()` indexing.

```
li = py.list({1,2,3,4});  
res = li(2:3)
```

```
res =
```

```
    Python list with no properties.
```

```
    [2.0, 3.0]
```

Use curly braces `{}` to get the contents of the element.

```
res = li{1}
```

```
res =
```

```
    1
```

Mapping Types

For mapping types, use curly braces with the Python key argument.

```
patient = py.dict(pyargs('name', 'John Doe', 'billing', 127));
patient{"billing"}
```

```
ans =
```

```
127
```

Size and Dimensions

MATLAB displays information for your system.

```
p = py.sys.path;
class(p)
```

```
ans =
```

```
py.list
```

Index into p.

```
p(1)
p{1}
```

```
ans =
```

```
Python list with no properties.
```

```
['c:\\work']
```

```
ans =
```

```
Python str with no properties.
```

```
c:\work
```

Inspect dimensions.

```
len = length(p)
sz = size(p)
```

```
len =  
    11  
  
sz =  
    1    11
```

Array Support

MATLAB converts a sequence type into a 1-by-N array.

Use Zero-Based Indexing for Python Functions

Python uses zero-based indexing; MATLAB uses one-based indexing. When you call a Python function, such as `py.sys.path`, the index value of the first element of a Python container, `x`, is `int32(0)`. The index value for the last element is `int32(py.Len(x) - 1)`.

See Also

Related Examples

- “Index into Python String” on page 20-34
- “Index into Python List” on page 20-35
- “Index into Python Tuple” on page 20-37
- “Index into Python dict” on page 20-38

More About

- “Explicit Type Conversions” on page 20-63
- “Limitations to Indexing into Python Objects” on page 20-74
- “Pass MATLAB Backslash Control Character” on page 20-16

View Python Numeric Values

In this section...

“Why Do I See Properties When I Display a Number?” on page 20-45

“What Is the L Character Attached to a Number?” on page 20-45

Why Do I See Properties When I Display a Number?

MATLAB displays all Python types as objects, including a list of properties of the object.

```
py.int(5)
ans =
    Python int with properties:
      denominator: 1
         imag: 0
      numerator: 5
         real: 5
    5
```

MATLAB displays the expected output value (5) on the last line.

What Is the L Character Attached to a Number?

Python appends an L character to the representation (display) of a long data type. For example, using Python version 2.7, type:

```
py.repr(py.long(5))
ans =
    Python str with no properties.
    5L
```

MATLAB displays Python `str` and appends L for any Python function that uses the `repr` function to display its output.

You treat a `long` data type like any numeric type. For example, add two numbers:

```
py.long(5) + py.long(2)
```

```
ans =
```

```
Python long with properties:
```

```
denominator: [1x1 py.long]  
  imag: [1x1 py.long]  
numerator: [1x1 py.long]  
  real: [1x1 py.long]
```

```
7
```

The answer is the number 7.

Call Methods on Python Variables

This example shows how to update a Python® list of folder names using the `append` method.

Create a list of folders `P` using the Python `sys.path` variable.

```
P = py.sys.path;
```

Display the Python functions for a `list` type.

```
methods(P)
```

```
Methods for class py.list:
```

```
append  count    ge        insert   lt        plus      reverse
cell    eq        gt        le       mtimes   pop       sort
char    extend   index    list     ne       remove
```

```
Methods of py.list inherited from handle.
```

Read the documentation for `append`.

```
py.help('list.append')
```

```
Help on method_descriptor in list:
```

```
list.append = append(...)
    L.append(object) -- append object to end
```

Add the current folder to the end of the path.

```
append(P, pwd)
```

Display the number of folders on the path. The list has `py.len` elements. Your value might be different. The type of this number is `py.int`.

```
py.len(P)
```

```
ans = int64
    10
```

See Also

Related Examples

- “Use Python list Type in MATLAB” on page 20-18

Reload Modified User-Defined Python Module

This example shows how to reload a modified Python module.

When you use this workflow, MATLAB deletes all variables, scripts, and classes in the workspace. For more information, see the `clear classes` function.

The Python calling syntax to reload the module depends on your Python version. To verify your Python version, use the MATLAB `pyenv` function.

Create Python Module

Change your current folder to a writable folder. Open a new file in MATLAB Editor.

Copy these statements defining a `myfunc` function and save the file as `mymod.py`.

```
def myfunc():  
    """Display message."""  
    return 'version 1'
```

Call `myfunc`.

```
py.mymod.myfunc
```

```
ans =
```

```
    Python str with no properties.
```

```
    version 1
```

Modify Module

Modify the function, replacing the `return` statement with the following:

```
    return 'version 2'
```

Save the file.

Unload Module

```
clear classes
```

MATLAB deletes all variables, scripts, and classes in the workspace.

Import Modified Module

```
mod = py.importlib.import_module('mymod');
```

Reload Module in Python Version 2.7

```
py.reload(mod);
```

Reload Module in Python Versions 3.x

```
py.importlib.reload(mod);
```

Call Function in Updated Module

Call the updated myfunc function.

```
py.mymod.myfunc
```

```
ans =
```

```
    Python str with no properties.
```

```
    version 2
```

See Also

clear | pyenv

Related Examples

- “System and Configuration Requirements” on page 20-51
- “Call User-Defined Python Module” on page 20-7

System and Configuration Requirements

In this section...

“Python Version Support” on page 20-51

“Set Python Version on Windows Platform” on page 20-51

“Set Python Version on Mac and Linux Platforms” on page 20-52

“Download 64-Bit Version of Python on Windows Platforms” on page 20-52

“Requirements for Building Python Executable” on page 20-52

Python Version Support

To call Python modules in MATLAB, you must have a supported version of the reference implementation (CPython) installed on your system. MATLAB supports versions 2.7, 3.6, and 3.7.

To determine if your system has a supported version, use the `pyenv` function. The value set by `pyenv` is persistent across MATLAB sessions. If you have multiple supported versions, use `pyenv` to display the default version used by MATLAB.

You cannot switch between versions of Python in a single MATLAB session. MATLAB automatically selects and loads a Python version when you type a Python command, such as:

```
py.funcname
```

If you want to change versions, restart MATLAB and then run `pyenv` with the new version information.

Set Python Version on Windows Platform

On Windows platforms, use either:

```
pyenv('Version','version')
```

or

```
pyenv('Version','executable')
```

Note If you downloaded a Python interpreter, but did not register it in the Windows registry, use:

```
pyenv('Version', 'executable')
```

Set Python Version on Mac and Linux Platforms

To set the version, type:

```
pyenv('Version', 'executable')
```

where *executable* is the full path to the Python executable file.

Download 64-Bit Version of Python on Windows Platforms

The architecture of Python must match the architecture of MATLAB. On the Python download website, downloads for Microsoft Windows platforms are 32-bit versions by default. To download the 64-bit version, choose options with the name "Windows x86-64 MSI installer".

Requirements for Building Python Executable

On Linux and Mac systems, if you build the Python executable, configure the build with the `--enable-shared` option.

See Also

pyenv

External Websites

- <https://www.python.org/downloads/>

Create a Python Object

The syntax to create a Python object `pyObj` is:

```
pyObj = py.modulename.ClassName(varargin)
```

where `varargin` is the list of constructor arguments specified by the `__init__` method in `ClassName`.

In MATLAB, Python objects are *reference types* (handle objects) and do not adhere to the MATLAB copy-on-assignment and pass-by-value rules. When you copy a handle object, only the handle is copied and both the old and new handles refer to the same data. When you copy a MATLAB variable (a value object), the variable data is also copied. The new variable is independent of changes to the original variable.

The following example creates an object of the `TextWrapper` class in the Python standard library `textwrap` module.

Read the constructor signature, `__init__`.

```
py.help('textwrap.TextWrapper.__init__')
```

```
Help on method __init__ in textwrap.TextWrapper:
```

```
textwrap.TextWrapper.__init__ = __init__(self, width=70, initial_indent='', subsequent_
```

Create a default `TextWrapper` object. You do not need to pass any input arguments because each argument has a default value, identified by the equal sign (=) character.

```
tw = py.textwrap.TextWrapper;
```

```
tw =
```

Python `TextWrapper` with properties:

```

        width: 70
    subsequent_indent: [1x1 py.str]
wordsep_simple_re_uni: [1x1 py._sre.SRE_Pattern]
    fix_sentence_endings: 0
        break_on_hyphens: 1
    break_long_words: 1
        wordsep_re_uni: [1x1 py._sre.SRE_Pattern]
        initial_indent: [1x1 py.str]
        expand_tabs: 1

```

```
replace_whitespace: 1
drop_whitespace: 1
```

```
<textwrap.TextWrapper instance at 0x000000006D58F808>
```

To change a logical value, for example, the `break_long_words` property, type:

```
tw.break_long_words = 0;
```

To change a numeric value, for example, the `width` property, first determine the numeric type.

```
class(tw.width)
```

```
ans =
```

```
int64
```

By default, when you pass a MATLAB number to a Python function, Python reads it as a float. If the function expects an integer, Python might throw an error or produce unexpected results. Explicitly convert the MATLAB number to an integer. For example, type:

```
tw.width = int64(3);
```

Read the help for the `wrap` method.

```
py.help('textwrap.TextWrapper.wrap')
```

```
Help on method wrap in textwrap.TextWrapper:
```

```
textwrap.TextWrapper.wrap = wrap(self, text) unbound textwrap.TextWrapper method
wrap(text : string) -> [string]
```

```
Reformat the single paragraph in 'text' so it fits in lines of
no more than 'self.width' columns, and return a list of wrapped
lines. Tabs in 'text' are expanded with string.expandtabs(),
and all other whitespace characters (including newline) are
converted to space.
```

Create a list of wrapped lines, `w`, from input `T`.

```
T = 'MATLAB® is a high-level language and interactive environment for numerical computa
w = wrap(tw,T);
whos w
```

Name	Size	Bytes	Class	Attributes
w	1x1	112	py.list	

Convert the `py.list` to a cell array and display the results.

```
wrapped = cellfun(@char, cell(w), 'UniformOutput', false);  
fprintf('%s\n', wrapped{:})
```

```
MATLAB®  
is  
a  
high-  
level  
language  
and  
interactive  
environment  
for  
numerical  
computation,  
visualization,  
and  
programming.
```

Although `width` is 3, setting the `break_long_words` property to `false` overrides the `width` value in the display.

See Also

Related Examples

- “Call Python from MATLAB” on page 20-4
- “Pass Keyword Arguments” on page 20-32

More About

- “Why Use Object-Oriented Design”

Pass Data to Python

In this section...

“MATLAB Type to Python Type Mapping” on page 20-56

“MATLAB Vector to Python Mapping” on page 20-57

“Passing Matrices and Multidimensional Arrays” on page 20-57

MATLAB Type to Python Type Mapping

When you pass MATLAB data as arguments to Python, MATLAB converts the data into types that best represent the data to the Python language.

MATLAB Input Argument Type — Scalar Values Only	Resulting Python py . Type
double single	float
Complex single Complex double	complex
int8 uint8 int16 uint16 int32	int
uint32 int64 uint64	int long (version 2.7 only)
NaN	float(nan)
Inf	float(inf)
string scalar char vector	str
<missing> value in string	None
logical	bool
Structure	dict

MATLAB Input Argument Type – Scalar Values Only	Resulting Python py . Type
Python object — <i>py.type</i>	<i>type</i>
function handle <i>@py.module.function</i> , to Python functions only	<i>module.function</i>

MATLAB Vector to Python Mapping

MATLAB Input Argument Type – 1-by-N Vector	Resulting Python Type
double	<code>array.array('d')</code>
single	<code>array.array('f')</code>
int8	<code>array.array('b')</code>
uint8	<code>array.array('B')</code>
int16	<code>array.array('h')</code>
uint16	<code>array.array('H')</code>
int32	<code>array.array('i')</code>
uint32	<code>array.array('I')</code>
int64 (Not supported for Python 2.7 on Windows)	<code>array.array('q')</code>
uint64 (Not supported for Python 2.7 on Windows)	<code>array.array('Q')</code>
char array containing values greater than 127 (version 2.7 only)	unicode
logical	memoryview
char vector string scalar	str
cell vector	tuple

Passing Matrices and Multidimensional Arrays

When you pass real numeric or logical arrays to a Python function, MATLAB automatically converts the data to a Python `memoryview` object. If the output of a Python function

implements the Python buffer protocol and is real numeric or logical, then MATLAB displays:

- The actual Python type
- The underlying data
- The corresponding MATLAB conversion function. Use this function to fully convert the Python object to a MATLAB array.

For example, suppose that you call a Python function in a module *pyModule* that returns a variable of type *pyType* with these values:

```
p =  
Python pyType:  
  
      8      1      6  
      3      5      7  
      4      9      2
```

Use `details` function to view the properties of the Python object.

Use `double` function to convert to a MATLAB array.

To convert `p` to a MATLAB matrix `P`, type:

```
P = double(p)
```

```
P = 3×3  
      8      1      6  
      3      5      7  
      4      9      2
```

If you need specific information about the Python properties of `p`, type:

```
details(p)
```

```
py.pyModule.pyType handle with properties:
```

```
      T: [1×1 py.pyModule.pyType]  
      base: [1×1 py.NoneType]  
      ctypes: [1×1 py.pyModule.core._internal._ctypes]  
      data: [1×3 py.memoryview]  
      dtype: [1×1 py.pyModule.dtype]  
      flags: [1×1 py.pyModule.flagsobj]  
      flat: [1×1 py.pyModule.flatiter]
```

```

    imag: [1x1 py.pyModule.pyType]
itemsize: [1x1 py.int]
nbytes: [1x1 py.int]
ndim: [1x1 py.int]
real: [1x1 py.pyModule.pyType]
shape: [1x2 py.tuple]
size: [1x1 py.int]
strides: [1x2 py.tuple]

```

Methods, Events, Superclasses

If the Python module provides content in its `__doc__` attribute, then MATLAB links to that information.

Using Python `memoryview` objects allows Python to read the MATLAB data without making a copy of the MATLAB data. For information about `memoryview` objects and buffer protocol, search for these terms at <https://www.python.org/doc/>.

MATLAB complex and sparse arrays are not supported in Python. See “Unsupported MATLAB Types” on page 20-62.

Troubleshooting Argument Errors

If a Python function expects a specific Python multidimensional array type, then MATLAB displays a message with tips about how to proceed. If the problem might be due to passing a matrix or a multidimensional array as an argument, then do the following.

- 1 Check the documentation for the Python function and find out the expected type for the argument.
- 2 Create a Python object of that type in MATLAB and pass that to the Python function.

For example, suppose that the following code returns an error.

```

a = [1 2; 3 4];
py.pyfunc(a)

```

If the documentation of `pyfunc` specifies that the expected type is `pyType`, then try this conversion:

```

py.pyfunc(pyType(a))

```

If the error persists, then determine the root cause by checking for additional information in the Python exception.

See Also

Related Examples

- “Call Python Methods with Numeric Arguments” on page 20-10
- “Pass MATLAB Character Vector to Python Method” on page 20-13
- “Create Python list Variable” on page 20-17
- “Create Python tuple Variable” on page 20-25
- “Create Python dict Variable” on page 20-27
- “Pass dict Argument to Python Method” on page 20-28
- “Pass Cell Array as Python Sequence Type” on page 20-22
- “Pass Keyword Arguments” on page 20-32

More About

- “Handle Data Returned from Python” on page 20-63
- “Unsupported MATLAB Types” on page 20-62

Troubleshooting Multidimensional Array Argument Errors

If a Python function expects a specific Python multidimensional array type, then MATLAB displays a message with tips about how to proceed. If the problem might be due to passing a matrix or a multidimensional array as an argument, then do the following.

- 1 Check the documentation for the Python function and find out the expected type for the argument.
- 2 Create a Python object of that type in MATLAB and pass that to the Python function.

For example, suppose that the following code returns an error.

```
a = [1 2; 3 4];  
py.pyfunc(a)
```

If the documentation of `pyfunc` specifies that the expected type is *pyType*, then try this conversion:

```
py.pyfunc(pyType(a))
```

If the error persists, then determine the root cause by checking for additional information in the Python exception.

See Also

Unsupported MATLAB Types

The following MATLAB types are not supported in Python.

- Multidimensional char or cell arrays
- Structure arrays
- Complex, scalar integers or arrays
- Sparse arrays
- `categorical`,
`table`,
`containers.Map`,
datetime types
- MATLAB objects
- `meta.class` (`py.class`)

See Also

More About

- “Pass Data to Python” on page 20-56

Handle Data Returned from Python

In this section...

“Automatic Python Type to MATLAB Type Mapping” on page 20-63

“Explicit Type Conversions” on page 20-63

Automatic Python Type to MATLAB Type Mapping

The following table shows how MATLAB converts data returned from Python into MATLAB types.

Python Return Type, as Displayed in Python	Resulting MATLAB Type — Scalar
<code>bool</code>	logical
<code>int</code> (version 2.7 only). For Python versions 3.x <code>int</code> , you must convert explicitly.	<code>int64</code>
<code>float</code>	double
<code>complex</code>	Complex double
All other Python types — <i>type</i>	Python object — <i>py.type</i>

Explicit Type Conversions

MATLAB provides the following functions to convert Python data types to MATLAB types manually.

Python Return Type or Protocol, as Displayed in MATLAB	MATLAB Conversion Function
<code>py.str</code> (version 2.7)	<code>string</code> <code>char</code> <code>uint8</code>
<code>py.str</code> (version 3.x)	<code>string</code> <code>char</code>
<code>py.unicode</code>	<code>string</code> <code>char</code>

Python Return Type or Protocol, as Displayed in MATLAB	MATLAB Conversion Function
Object with <code>__str__</code> method	<code>char</code>
<code>py.bytes</code>	<code>uint8</code>
<code>py.int</code>	<code>double</code> or <code>int64</code>
<code>py.long</code>	<code>double</code> or <code>int64</code>
<code>py.array.array</code> ^a	<code>numeric</code> <code>double</code> <code>single</code> <code>int8</code> <code>uint8</code> <code>int16</code> <code>uint16</code> <code>int32</code> <code>uint32</code> <code>int64</code> <code>uint64</code>
Sequence protocol; for example, <code>py.list</code> and <code>py.tuple</code>	<code>cell</code>
Mapping protocol; for example, <code>py.dict</code>	<code>struct</code>

a. You can convert `py.array.array` of any format to the MATLAB type you want.

See Also

More About

- “Use Python list Type in MATLAB” on page 20-18
- “Use Python dict Type in MATLAB” on page 20-29
- “Convert Python dict Type to MATLAB Structure” on page 20-31
- “Use Python Numeric Types in MATLAB” on page 20-9

- “Pass Data to Python” on page 20-56

How MATLAB Represents Python Operators

MATLAB supports the following overloaded operators.

Python Operator Symbol	Python Methods	MATLAB Methods
+ (binary)	<code>__add__</code> , <code>__radd__</code>	plus, +
- (binary)	<code>__sub__</code> , <code>__rsub__</code>	minus, -
* (binary)	<code>__mul__</code> , <code>__rmul__</code>	mtimes, *
/	<code>__truediv__</code> , <code>__rtruediv__</code>	mrdivide, /
==	<code>__eq__</code>	eq, ==
>	<code>__gt__</code>	gt, >
<	<code>__lt__</code>	lt, <
!=	<code>__ne__</code>	ne, ~=
>=	<code>__ge__</code>	ge, >=
<=	<code>__le__</code>	le, <=
- (unary)	<code>__neg__</code>	uminus, -a
+ (unary)	<code>__pos__</code>	uplus, +a

The following Python operators are not supported.

Python Operator Symbol	Python Method
%	<code>__mod__</code> , <code>__rmod__</code>
**	<code>__pow__</code> , <code>__rpow__</code>
<<	<code>__lshift__</code> , <code>__rlshift__</code>
>>	<code>__rshift__</code> , <code>__rrshift__</code>
&	<code>__and__</code> , <code>__rand__</code>
^	<code>__xor__</code> , <code>__rxor__</code>
	<code>__or__</code> , <code>__ror__</code>
// (binary)	<code>__floordiv__</code> , <code>__rfloordiv__</code>
+= (unary)	<code>__iadd__</code>

Python Operator Symbol	Python Method
-= (unary)	<code>__isub__</code>
*= (unary)	<code>__imul__</code>
/= (unary)	<code>__itruediv__</code>
//= (unary)	<code>__ifloordiv__</code>
%= (unary)	<code>__imod__</code>
**= (unary)	<code>__ipow__</code>
<<= (unary)	<code>__ilshift__</code>
>>= (unary)	<code>__irshift__</code>
&= (unary)	<code>__iand__</code>
^= (unary)	<code>__ixor__</code>
!= (unary)	<code>__ior__</code>
~ (unary)	<code>__invert__</code>

Execute Callable Python Object

To execute a callable Python object, use the `feval` function. For example, if instance *obj* of a Python class is callable, replace the Python syntax `obj(x1, ..., xn)` with one of the following MATLAB statements:

```
feval(obj,x1, ..., xn)
```

```
obj(x1, ..., xn)
```

See Also

`feval`

Understanding Python and MATLAB import Commands

The `import` statement does not have the same functionality in MATLAB as in Python. Python uses the `import` statement to load and make code accessible. MATLAB uses the `import` function to refer to a class or function without using the package name.

Caution Never call:

```
import py.*
```

If you do, then MATLAB calls the Python function instead of the MATLAB function of the same name. This can cause unexpected behavior.

If you call this command, then you must call the MATLAB command:

```
clear import
```

Do Not Type `import pythonmodule`

MATLAB automatically loads Python when you type:

```
py.command
```

Do not type in MATLAB:

```
import pythonmodule
```

Use MATLAB `import` to Shorten Class or Function Names

The Python `from . . . import` statement lets you reference a module without using the fully qualified name. Suppose that you have the following Python code, where `y` is a class or function name that you want to use.

```
from x import y
```

Replace this statement with the following MATLAB code:

```
import x.y
```

For example, the Python `textwrap` module formats blocks of text.

```
S = py.textwrap.wrap('This is a string');
```

Since `wrap` is not a MATLAB function, you can shorten the calling syntax using the `import` function. After calling this command, you do not need to type the package (`py`) and module (`textwrap`) names.

```
import py.textwrap.wrap  
S = wrap('This is a string');
```

See Also

`import`

List, Tuple, and Dictionary Types

The following table shows the commands for creating list, tuple, and dict types. The commands on the left are run from the Python interpreter. The commands on the right are MATLAB commands.

Python list – []	MATLAB py.list
['Robert', 'Mary', 'Joseph']	py.list({'Robert', 'Mary', 'Joseph'})
[[1,2],[3,4]]	py.list({py.list([1,2]),py.list([3,4])})
Python tuple – ()	MATLAB py.tuple
('Robert', 19, 'Biology')	py.tuple({'Robert',19,'Biology'})
Python dict – {}	MATLAB py.dict
{'Robert': 357, 'Joe': 391, 'Mary': 229}	py.dict(pyargs(... 'Robert',357,'Mary',229,'Joe',391))

See Also

pyargs

Related Examples

- “Use Python list Type in MATLAB” on page 20-18
- “Create Python tuple Variable” on page 20-25
- “Use Python dict Type in MATLAB” on page 20-29

Limitations to Python Support

Features Not Supported in MATLAB
Closing the Python interpreter while running MATLAB.
Saving (serializing) Python objects into a MAT-file.
Interactive Python help (calling <code>py.help</code> without input arguments).
<code>py.input</code> and <code>py.raw_input</code> (version 2.7).
Accessing static properties of a Python class.
MATLAB <code>isa</code> function does not recognize virtual inheritance.
MATLAB class inheritance from a Python class.
Customized (dynamic) attribute access.
Nested Python classes.
Modules that start MATLAB in a separate process, for example, the <code>multiprocessing</code> module.
Modules that read <code>sys.argv</code> , the command-line arguments passed to a Python script, for example, <code>Tkinter</code> .
Dynamically generated Python classes, for example, <code>collections.namedtuple</code> in CPython 2.7.
Dynamically attaching new object attributes. Instead, use <code>py setattr</code> .
Class names or other identifiers starting with an underscore (<code>_</code>) character. Instead, use the Python <code>py getattr</code> and <code>py setattr</code> functions.
Python modules generated by the MATLAB Compiler SDK product.
Python code using Cocoa (AppKit) for user interfaces on macOS platforms.
The size of variables passed between Python and MATLAB is limited to 2 GB when you call a Python function out-of-process. This limit applies to the data plus supporting information passed between the processes.

See Also

More About

- “Understanding Python and MATLAB import Commands” on page 20-69
- “Unsupported MATLAB Types” on page 20-62

Limitations to Indexing into Python Objects

You can access data in Python container objects, like lists and dictionaries, with index values, similar to referencing an element in a MATLAB matrix. There are, however, ways to index into matrices which are not supported for these Python types.

Indexing Features Not Supported in MATLAB
Use of square brackets, [].
Indexing into a container type that does not inherit from <code>collections.Sequence</code> or <code>collections.Mapping</code> .
Logical indexing.
Accessing data in a container with an arbitrary array of indices. An index must be of the form <code>start:step:stop</code> .
Comma-separated lists.
<code>numel</code> function does not return number of array elements. Returns 1.

See Also

More About

- “Array Indexing”

Undefined variable "py" or function "py.command"

MATLAB automatically loads Python when you type `py.command` in the Command Window. If MATLAB displays this message, a failure has occurred.

```
Undefined variable "py" or function "py.command"
```

Use this page to help troubleshoot the failure.

Python Not Installed

A supported version of Python is not installed on your computer. Review “Call Python Library Functions” for your MATLAB version, then download and install Python from <https://www.python.org/downloads/>.

On Microsoft Windows, Python downloads the 32-bit version of Python by default. If you run a 64-bit version of MATLAB, then download a 64-bit version of Python, identified by the name "Windows x86-64 MSI installer". For more information, see “Download 64-Bit Version of Python on Windows Platforms” on page 20-52.

On Linux and Mac systems, if you build Python from source files, then configure the build with the `--enable-shared` option.

64-bit/32-bit Versions of Python on Windows Platforms

You installed a 32-bit version of Python for a 64-bit version of MATLAB. You must install a 64-bit version of Python.

MATLAB Cannot Find Python

Python is in a nonstandard location. To provide the path to the Python executable, use the `pyenv` function. For example:

```
pyenv('Version','C:\Users\uname\WinPython-64bit-3.2.1\python-3.2.1.amd64\python.exe')
```

On Windows systems, Python is not found in the Windows registry. If you downloaded a Python interpreter, but did not register it in the Windows registry, specify the Python location using the command:

```
pyenv('Version','executable')
```

Error in User-Defined Python Module

An error occurred in the user-defined Python module. To test if your module, *mymod*, contains errors, type:

```
py.importlib.import_module('mymod')
```

If Python detects an error in the module, then MATLAB displays a Python error message.

Alternatively, execute the equivalent command at the Python command prompt to get the Python error message.

After you fix the error, to access the updated module, restart MATLAB, and add it to the search path.

Python Module Not on Python Search Path

If *command* is a valid Python command, make sure the Python module is on the Python search path. To test if module *mymod* is on the path, type:

```
py.importlib.import_module('mymod')
```

If Python cannot find the module, MATLAB displays a Python error message.

To add *mymod*, in folder *modpath*, to the path, type:

```
P = py.sys.path;  
if count(P,'modpath') == 0  
    insert(P,int32(0),'modpath');  
end
```

The Python search path is associated with the Python interpreter loaded in the current session of MATLAB. You can modify the search path in MATLAB, but the modifications are not present if you run other instances of the interpreter outside of MATLAB.

Module Name Conflicts

If you call a Python module that has the same name as a module in the standard library or any 3rd-party modules installed on your system, then MATLAB might load the wrong module.

Python Tries to Execute command in Wrong Module

If *command* is in a user-defined module, then make sure that the module name does not conflict with modules in the Python standard library or any 3rd-party modules on your system.

See Also

pyenv

More About

- “System and Configuration Requirements” on page 20-51

External Websites

- <https://www.python.org/downloads/>

Help for Python Functions

For a complete description of Python functionality, consult outside resources, in particular, `python.org`. There are different versions of the Python documentation, so be sure to refer to the version corresponding to the version on your system. Many examples in the MATLAB documentation refer to functions in the Python standard library.

To use functions in a third-party or user-defined Python module, refer to your vendor product documentation for information about how to install the module and for details about its functionality.

The `py.help` command displays the Python help found at `www.python.org/doc`. Help for packages and classes can be extensive and might not be useful when displayed in the MATLAB command window.

- Package

```
py.help('textwrap')
```

- Class

```
py.help('textwrap.TextWrapper')
```

- Method of a class

```
py.help('textwrap.TextWrapper.wrap')
```

- Function

```
py.help('textwrap.fill')
```

If MATLAB displays an error message beginning with `Python Error:`, refer to your Python documentation for more information.

Note You cannot use the interactive Python help, calling `py.help` without input arguments, in MATLAB.

See Also

More About

- “Handle Python Exceptions” on page 20-80

External Websites

- www.python.org

Handle Python Exceptions

MATLAB catches exceptions thrown by Python and converts them into a `matlab.exception.PyException` object, which is derived from the `MException` class. For example:

```
try
    py.list('x','y',1)
catch e
    e.message
    if(isa(e,'matlab.exception.PyException'))
        e.ExceptionObject
    end
end

ans =

Python Error: TypeError: list() takes at most 1 argument (3 given)

ans =

    Python tuple with no properties.

    (<type 'exceptions.TypeError'>, TypeError('list() takes at most 1 argument (3 given)'), None)
```

If MATLAB displays an error message of the following format, refer to your Python documentation for more information.

Python Error: *Python class: message*

See Also

`matlab.exception.PyException`

Troubleshooting Error Messages

Troubleshooting errors when using a MATLAB external interface is a challenge. Is the error in the Python application or in your MATLAB code? Common errors include errors reported by Python and errors from attempting to convert Python data to MATLAB and conversely.

Python Error: Python class: message

MATLAB displays an error message in the following format:

```
Python Error: Python class: message
```

MATLAB displays *message* only if there is a Python error message.

This error comes from Python and for information you must refer to your version of Python documentation at www.python.org/doc or the product documentation from third-party vendors. For example:

```
p = py.os.path.split(pwd);  
py.operator.setitem(p,int32(1),py.str('temp'));
```

```
Python Error: TypeError: 'tuple' object does not support item assignment
```

Search for the term “tuple” on the Python documentation site for your version of Python. Tuple is a built-in function described here: <https://docs.python.org/2/library/functions.html#tuple>.

Python Module Errors

MATLAB reports some Python errors as a MATLAB error loading a module. For more information, see “Undefined variable “py” or function “py.command”” on page 20-75.

If you write your own Python modules or modify the source code from an existing module, test your MATLAB commands by writing the equivalent Python command in your Python interpreter. This workflow is beyond the scope of MATLAB documentation and product support.

Errors Converting Python Data

When the data is compatible, MATLAB automatically converts Python data to MATLAB data. For the list of data types you must explicitly convert, see “Explicit Type Conversions” on page 20-63.

See Also

More About

- “Python Data Types in MATLAB”
- “Limitations to Python Support” on page 20-72

External Websites

- www.python.org/doc

Using Python Data in MATLAB

MATLAB automatically converts compatible Python data to MATLAB data.

For Python types that do not have compatible MATLAB types, such as `list` and `dict`, use the MATLAB functions shown in the Explicit Type Conversion table. This table also describes how to convert strings and certain numeric types explicitly.

See Also

More About

- “Automatic Python Type to MATLAB Type Mapping” on page 20-63
- “Explicit Type Conversions” on page 20-63

Call Python eval Function

This example shows how to evaluate the expression `x+y` in Python®. To evaluate an expression, pass a Python dict value for the `globals` namespace parameter.

Read the help for `eval`.

```
py.help('eval')
```

```
Help on built-in function eval in module __builtin__:
```

```
eval(...)  
    eval(source[, globals[, locals]]) -> value
```

```
    Evaluate the source in the context of globals and locals.  
    The source may be a string representing a Python expression  
    or a code object as returned by compile().  
    The globals must be a dictionary and locals can be any mapping,  
    defaulting to the current globals and locals.  
    If only globals is given, locals defaults to it.
```

Create a Python dict variable for the `x` and `y` values.

```
workspace = py.dict(pyargs('x',1,'y',6))
```

```
workspace =  
    Python dict with no properties.
```

```
    {'y': 6.0, 'x': 1.0}
```

Evaluate the expression.

```
res = py.eval('x+y',workspace)
```

```
res = 7
```

Add two numbers without assigning variables. Pass an empty dict value for the `globals` parameter.

```
res = py.eval('1+6',py.dict)
```

```
res = int64  
    7
```


Precedence Order of Methods and Functions

If a Python class defines a method with the same name as a MATLAB converter method for Python types, MATLAB calls the Python method. This means you cannot call the MATLAB converter method on an object of that class.

For example, if a Python class defines a `char` method, the following statement calls the Python method.

```
char(obj)
```

To use the MATLAB `char` function, type:

```
char(py.str(obj))
```


Python Function Arguments

In this section...

“Positional Arguments” on page 20-87

“Keyword Arguments” on page 20-87

“Optional Arguments” on page 20-88

Your Python documentation shows you how to call a Python function. Python function signatures look similar to MATLAB function signatures. However, Python has syntax which might be unfamiliar to MATLAB users.

Positional Arguments

A positional argument is passed by position. These arguments appear at the beginning of a function signature.

Python Signature	MATLAB Usage
<code>abs(X)</code> Argument X is required.	<code>py.abs(-99)</code>

Some functions accept an arbitrary sequence of positional arguments, including no arguments. In Python, these arguments are defined by prepending the name with the * character.

Python Signature	MATLAB Usage
<code>itertools.izip(*iterables)</code> The <code>iterables</code> argument is not required, in which case, the function returns a zero length iterator.	Aggregate elements from two lists. <code>py.itertools.izip(... py.list({1:10}),py.list({'a','b'})) ;</code> Create zero length iterator. <code>py.itertools.izip;</code>

Keyword Arguments

A keyword argument is preceded by an identifier. Keyword arguments, also called named arguments, can be specified in any order.

Keyword arguments are like name-value pairs in MATLAB. Use the MATLAB `pyargs` function to create keyword arguments for Python functions.

Python Signature	MATLAB Usage
<pre>print(*objects, sep=' ', end='\n', file=sys.stdout) sep, end, and file are keyword arguments.</pre>	Change the value of end. <pre>py.print('string', pyargs('end', '--'))</pre>

Python defines an arbitrary number of keyword arguments by prepending the name with `**` characters.

Python Signature	MATLAB Usage
<pre>dict(**kwarg)</pre>	<pre>D = py.dict(pyargs('Joe', 100, 'Jack', 101))</pre>

Optional Arguments

An optional argument is a non-required argument.

Python Signature	MATLAB Usage
<pre>random.randrange(start, stop[, step])</pre> Argument <code>step</code> is optional.	<pre>py.random.randrange(1, 100)</pre>

Optional arguments can have default values. A default value is indicated by an equal sign `=` with the default value.

Python Signature	MATLAB Usage
<pre>print(*objects, sep=' ', end='\n', file=sys.stdout)</pre> The default value for <code>file</code> is <code>sys.stdout</code> .	Print two values using default keyword values. <pre>py.print(2, '2')</pre>

See Also

`pyargs`

Related Examples

- “Pass Keyword Arguments” on page 20-32

Out-of-Process Execution of Python Functionality

Note There is overhead associated with calling Python functions out-of-process. This behavior affects performance. MathWorks recommends calling Python functions in-process, which is the default mode.

MATLAB can run Python scripts and functions in a separate process. Running Python in a separate process enables you to:

- Use some third-party libraries in the Python code that are not compatible with MATLAB.
- Isolate the MATLAB process from crashes in the Python code.

To run out-of-process, call the `pyenv` function with the `"ExecutionMode"` argument set to `"OutOfProcess"`. For example, suppose that you want to create this `list` variable in the Python environment.

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

To create this `list` out-of-process, set the MATLAB execution mode to `"OutOfProcess"`. MATLAB displays information about your current Python environment.

```
pyenv("ExecutionMode", "OutOfProcess")  
  
ans =  
  PythonEnvironment with properties:  
  
    Version: "2.7"  
  Executable: "C:\Python27\pythonw.exe"  
    Library: "C:\windows\system32\python27.dll"  
      Home: "C:\Python27"  
    Status: NotLoaded  
  ExecutionMode: OutOfProcess
```

Create the variable.

```
py.list({'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'})  
  
ans =  
  
  Python list with no properties.
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

MATLAB creates a process named `MATLABPyHost`.

```
pyenv
```

```
ans =
```

```
PythonEnvironment with properties:
```

```
Version: "2.7"  
Executable: "C:\Python27\pythonw.exe"  
Library: "C:\windows\system32\python27.dll"  
Home: "C:\Python27"  
Status: Loaded  
ExecutionMode: OutOfProcess  
ProcessID: "8196"  
ProcessName: "MATLABPyHost"
```

Note Clearing a Python object is asynchronous, which means that the Python object might remain in Python after the invocation of a synchronous call. For example, in the following code it is possible that `myList2` is created before `myList` is destroyed.

```
myList=py.list;  
clear myList  
myList2 = py.list;
```

Limitations

The size of variables passed between Python and MATLAB is limited to 2 GB when you call a Python function out-of-process. This limit applies to the data plus supporting information passed between the processes.

See Also

```
pyenv
```


System Commands

Shell Escape Function Example

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command `!` to make external stand-alone programs act like new MATLAB functions.

For example, the following function, `garfield.m`, uses an external function, `gareqn`, to find the solution to Garfield's equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This function:

- 1 Saves the input arguments `a`, `b`, `q`, and `r` to a MAT-file in the workspace using the `save` command.
- 2 Uses the shell escape operator to access a C or Fortran program called `gareqn` that uses the workspace variables to perform its computation. `gareqn` writes its results to the `gardata` MAT-file.
- 3 Loads the `gardata` MAT-file to obtain the results.

See Also

Related Examples

- “Run External Commands, Scripts, and Programs” on page 21-3

Run External Commands, Scripts, and Programs

You can execute operating system commands from the MATLAB command line using the `!` operator or the `system` function.

Shell Escape Function

The exclamation point character (`!`), sometimes called *bang*, is a *shell escape*. The `!` character indicates that the rest of the input line is a command to the operating system. The operating system determines the maximum length of the argument list you can provide as input to the command. Use `!` to call utilities or other executable programs without quitting MATLAB.

For example, the following code opens the vi editor for a file named `yearlystats.m` on a UNIX platform.

```
!vi yearlystats.m
```

After the external program completes or you quit the program, the operating system returns control to MATLAB. To run the application in background mode or display the output in a separate window, add `&` to the end of the line.

For example, the following statement opens the Microsoft Excel program and returns control to the command prompt so that you can continue running MATLAB commands.

```
!excel.exe &
```

The following command on a Windows platform displays the results in a DOS window.

```
!dir &
```

Note To use the exclamation point in a factorial expression, call the `factorial` function.

Return Results and Status

To run a program that returns results and status, use the `system` function.

Specify Environment Variables

To execute operating system commands with specific environment variables, include all commands to the operating system within the system call. This applies to the MATLAB ! (bang), system, dos, and unix functions. To separate commands:

- On Windows platforms, use & (ampersand)
- On UNIX platforms, use ; (semicolon)

Alternatively, set environment variables before starting MATLAB.

Run UNIX Programs off System Path

You can run a UNIX program from MATLAB when the folder containing that file is not on the UNIX system path that is visible to MATLAB. To view the path visible to MATLAB, type the following at the MATLAB command prompt.

```
getenv('PATH')
```

You can modify the system path for the current MATLAB session or across subsequent MATLAB sessions, as described in these topics:

- “Current MATLAB Session” on page 21-4
- “Across MATLAB Sessions Within Current Shell Session” on page 21-5
- “Across All MATLAB Sessions” on page 21-5

Current MATLAB Session

You can modify the system path for the current MATLAB session. When you restart MATLAB, the folder is no longer on the system path.

To modify the system path, do one of the following.

- Change the current folder in MATLAB to the folder that contains the program you want to run.
- Type the following commands at the command prompt.

```
path1 = getenv('PATH')
path1 = [path1 ':/usr/local/bin']
setenv('PATH', path1)
!echo $PATH
```

Across MATLAB Sessions Within Current Shell Session

You can modify the system path within a shell session. When you restart MATLAB within the current shell session, the folder remains on the system path. However, if you restart the shell session, and then restart MATLAB, the folder is no longer on the path.

To add a folder to the system path from the shell, do the following.

- 1 Exit MATLAB.
- 2 Depending on the shell you are using, type one of the following at the system command prompt, where *myfolder* is the folder that contains the program you want to run:
 - For bash or related shell:


```
export PATH="$PATH:myfolder"
```
 - For tcsh or related shell:


```
setenv PATH "${PATH}:myfolder"
```
- 3 Start MATLAB.
- 4 In the MATLAB Command Window, type:

```
!echo $PATH
```

Across All MATLAB Sessions

To modify the system path across shell and MATLAB sessions, add the following commands to the MATLAB startup file as described in “Startup Options in MATLAB Startup File”.

```
path1 = getenv('PATH')
path1 = [path1 ':/usr/local/bin']
setenv('PATH', path1)
!echo $PATH
```

Run AppleScript on macOS

On macOS platforms, you cannot run the Apple AppleScript program directly from MATLAB. To run AppleScript commands, call the Apple macOS `osascript` function using the MATLAB `unix` or `!` (bang) functions.

See Also
system

Change Environment Variable for Shell Command

This example shows how to substitute a user-specified value for an environment variable value set by MATLAB when you call a function using the `system` command.

When you use the `system` command to call a function, the function inherits the MATLAB environment. To change environment variable values, use a shell wrapper. Use the environment variable `MATLAB_SHELL` to specify a shell containing your updated variable. This example uses a custom value for the environment variable `LD_LIBRARY_PATH`.

Create a wrapper file `matlab_shell.sh` in the folder `<PATH_TO_SHELL_SCRIPT>` with the following contents, where `<MY_LIBRARY_PATH>` is your custom value.

```
#!/bin/sh

LD_LIBRARY_PATH=<MY_LIBRARY_PATH>
export LD_LIBRARY_PATH

exec ${SHELL:-/bin/sh} $*
```

If you have a user-defined value for `SHELL`, the expression `${SHELL:-/bin/sh}` uses your `SHELL` value. Otherwise, MATLAB uses the Bourne shell.

From the operating system prompt, call MATLAB setting `MATLAB_SHELL` to:

```
<PATH_TO_SHELL_SCRIPT>/matlab_shell.sh
```

Display your value of `LD_LIBRARY_PATH` from the MATLAB command prompt.

```
!echo $LD_LIBRARY_PATH
```

Now when you call a function using the `system` command, the function uses the `LD_LIBRARY_PATH` value specified by `<MY_LIBRARY_PATH>`.

See Also

`system`

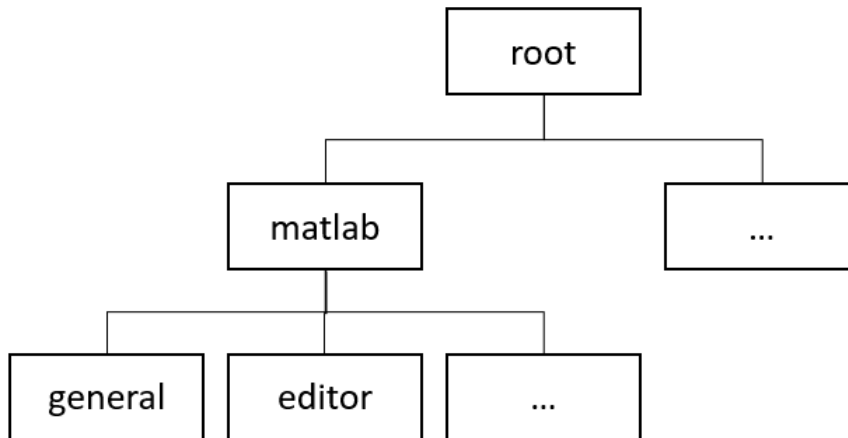
Settings

Access and Modify Settings

Settings provide a way to programmatically access and modify options for tools. For example, you can use settings to customize the appearance and behavior of the MATLAB editor, change the code font used by MATLAB desktop tools, or change how MAT-files are saved. Settings can be changed for the current session using temporary values, or across multiple sessions using personal values. For documentation on individual settings, go to “System Commands” and select a link in the **Settings** category.

Access Settings

Settings are organized by product in a tree-based hierarchy of settings groups. At the top of the tree is the root settings group object. Directly under the root object are the product settings groups. Each product settings group then contains its own hierarchy of settings. The leaf nodes in the settings tree are known as the settings.



To access a setting, use the `settings` function to get the root of the settings tree.

```
s = settings
```

Use dot notation to access the settings groups and settings in the tree. For example, view the list of settings groups in MATLAB.

```
s.matlab
```



```
ans =

SettingsGroup 'matlab' with properties:

    toolboxpathcache: [1x1 SettingsGroup]
      appdesigner: [1x1 SettingsGroup]
        editor: [1x1 SettingsGroup]
      general: [1x1 SettingsGroup]
        fonts: [1x1 SettingsGroup]
```

To get the current value for a setting, type the entire setting name using dot notation, including parent settings groups. For example, get the list of values for the maximum column width for comments in MATLAB.

```
s.matlab.editor.language.matlab.comments.MaxWidth

ans =

Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

    ActiveValue: 75
  TemporaryValue: <no value>
    PersonalValue: <no value>
    FactoryValue: 75
```

Modify Settings

A setting has four value types.

- **Active** — The active value is the current value of the setting.
- **Temporary** — The temporary value is available only for the current MATLAB session and is cleared at the end of the session.
- **Personal** — The personal value is persistent across MATLAB sessions for an individual user. When modified, the value is saved to the preferences folder.
- **Factory** — The factory value is the default setting value.

The active value of a setting is determined as follows:

- If the setting has a temporary value, then the active value is the temporary value.
- If the setting has no temporary value, but it has a personal value, then the active value is the personal value.
- If the setting has no temporary value or personal value, then the active value is the factory value.

For example, suppose you have a setting `MySetting` with a temporary value of 12, a factory value of 10, and no personal value. In this case, the active value for `MySetting` is the temporary value, 12.

To change the active value for a setting, set either the temporary or personal value for the setting. For example, set the temporary value for the maximum column width for comments in MATLAB to 80. This temporary value will be cleared at the end of the current MATLAB session.

```
s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80
s.matlab.editor.language.matlab.comments.MaxWidth

ans =

Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

    ActiveValue: 80
  TemporaryValue: 80
 PersonalValue: <no value>
   FactoryValue: 75
```

Restore Default Values

To restore the default value of setting, clear the temporary or personal values for the setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example, clear the temporary value for the maximum column width for comments in MATLAB. Use the `hasTemporaryValue` function to check whether the value exists before clearing it. Since the personal value for the setting is not defined, the factory value becomes the active value.

```
if(hasTemporaryValue(s.matlab.editor.language.matlab.comments.MaxWidth))
    clearTemporaryValue(s.matlab.editor.language.matlab.comments.MaxWidth)
end

s.matlab.editor.language.matlab.comments.MaxWidth

ans =

Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

    ActiveValue: 75
  TemporaryValue: <no value>
 PersonalValue: <no value>
   FactoryValue: 75
```

Settings and Preferences

Some settings are linked to a corresponding preference. If a setting is linked to a preference, changing the temporary or personal value for a setting changes the corresponding preference. If the temporary value is changed, the preference regains its

original value at the end of the MATLAB session. For more information about preferences, see “Preferences”.

See Also

settings

More About

- “Preferences”
- “Create Custom Settings” on page 22-6

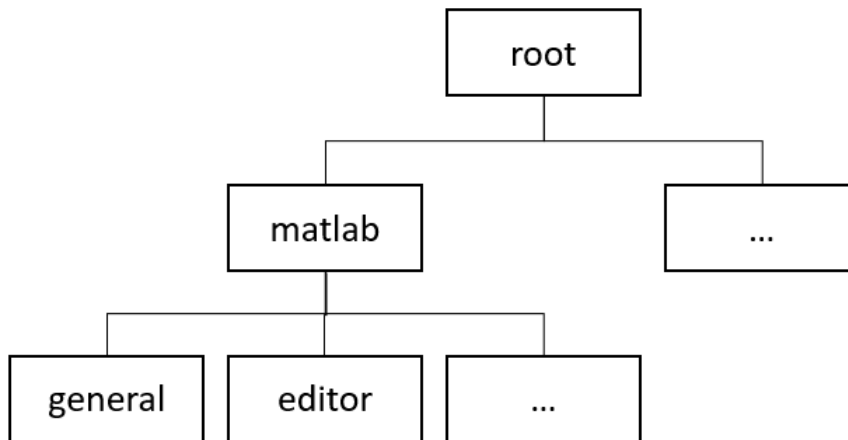
Create Custom Settings

Settings provide a way to programmatically store, access, and modify data for the current session or across multiple sessions. By default, MATLAB and other MathWorks products include settings that can be used to access and modify the appearance and behavior of tools. For example, MATLAB includes settings that enable you to programmatically change the font for certain code tools.

You can create custom settings to store and access your own data across sessions. For example, you can create settings to store the location of an important folder on your system or to keep track of the number of times a file is run.

Add and Remove Settings Groups

Settings are organized into groups. Grouping settings makes it easier to find a specific setting and also provides additional context as to what the setting is used for. For instance, the `matlab.editor` settings group contains all of the settings specific to the MATLAB Editor. Settings groups are organized into larger groups, forming a tree. At the top of the tree is the root settings group object.



To add a new settings group, use the `addGroup` function. For example, create the settings group `mysettings` under the root settings group object.

```
s = settings;
addGroup(s, 'mysettings');
s

s =
  SettingsGroup with properties:
      matlab: [1x1 SettingsGroup]
    mysettings: [1x1 SettingsGroup]
  mldrivetriplewireaccess: [1x1 SettingsGroup]
```

To create a hidden settings group that does not appear in the settings hierarchy, for example when you display the parent settings group, specify the 'Hidden' name-value pair. Hidden settings groups do not appear in the parent settings group but can be accessed programmatically. For example, create the settings group `myhiddensettings` inside `mysettings`. Notice that `myhiddensettings` does not display in `mysettings`.

```
addGroup(s.mysettings, 'myhiddensettings', 'Hidden', true);
s.mysettings
```

```
ans =
  SettingsGroup 'mysettings' with no properties.
```

To remove a settings group, use the `removeGroup` function. For example, remove `myhiddensettings`.

```
removeGroup(s, 'myhiddensettings');
```

Add and Remove Settings

To add a new setting, use the `addSetting` function. For example, add the setting `MyWorkAddress` to the `mysettings` settings group.

```
s = settings;
addGroup(s, 'mysettings');
addSetting(s.mysettings, 'MyWorkAddress');
```

Note Adding settings directly to the root settings group is not supported.

Once you have created a setting, you can give it a value. A setting has several different value types that let you set the value for just the current session or across multiple sessions for an individual user. For more information about these types of values, see “Access and Modify Settings” on page 22-2.

To specify a value for the setting, set its personal or temporary value. You cannot specify the factory value for a custom setting. For example, specify a personal value for `MyWorkAddress`.

```
s.mysettings.MyWorkAddress.PersonalValue = '3 Apple Hill Drive';
```

You then can use the setting value programmatically in your code.

```
fprintf('I work at %s.\n', s.mysettings.MyWorkAddress.ActiveValue)
```

```
I work at 3 Apple Hill Drive.
```

To add a hidden setting, use the `'Hidden'` name-value pair argument. Hidden settings do not display in the settings hierarchy, for example when you display the parent settings group, but are accessible programmatically. For example, add the hidden setting `MyHiddenWorkAddress` to the `mysettings` settings group and set its personal value.

```
addSetting(s.mysettings, 'MyHiddenWorkAddress', 'Hidden', true, ...
    'PersonalValue', '1 Lakeside Campus Drive');
```

You also can add read-only settings using the `'ReadOnly'` name-value pair argument. Once you create a read-only setting, you cannot change its temporary or personal value. Therefore, you must specify the personal value for the setting when you add it. For example, add the read-only setting `MyBirthDate` to the `mysettings` settings group with a specified personal value.

```
mydate = datetime('6/1/1990', 'InputFormat', 'MM/dd/yyyy');
addSetting(s.mysettings, 'MyBirthDate', 'ReadOnly', true, 'PersonalValue', mydate);
```

Validate Settings Using Functions

You can impose specific restrictions on settings values by specifying a validation function for a setting or group. A validation function accepts a potential setting value as an argument, and throws an error if the value does not meet a specific requirement.

MATLAB defines several useful validation functions that can be used to validate settings. This table lists these functions, their meanings, and the MATLAB functions they use.

Name	Meaning	Functions Called on Inputs
<code>matlab.settings.mustBeStringScalar(A)</code>	A is a string array with one element.	<code>isStringScalar</code>

Name	Meaning	Functions Called on Inputs
<code>matlab.settings.mustBeLogicalScalar(A)</code>	A is logical with one element.	<code>islogical</code> , <code>isscalar</code>
<code>matlab.settings.mustBeNumericScalar(A)</code>	A is numeric with one element.	<code>isnumeric</code> , <code>isscalar</code>
<code>matlab.settings.mustBeIntegerScalar(A)</code>	A is an integer with one element.	<code>isinteger</code> , <code>isscalar</code>
<code>mustBePositive(A)</code>	$A > 0$	<code>gt</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNonpositive(A)</code>	$A \leq 0$	<code>ge</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeFinite(A)</code>	A has no NaN and no Inf elements.	<code>isfinite</code>
<code>mustBeNonNan(A)</code>	A has no NaN elements.	<code>isnan</code>
<code>mustBeNonnegative(A)</code>	$A \geq 0$	<code>ge</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNegative(A)</code>	$A < 0$	<code>lt</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNonzero(A)</code>	$A \neq 0$	<code>eq</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNonempty(A)</code>	A is not empty.	<code>isempty</code>
<code>mustBeNonsparse(A)</code>	A has no sparse elements.	<code>issparse</code>
<code>mustBeNumeric(A)</code>	A is numeric.	<code>isnumeric</code>
<code>mustBeNumericOrLogical(A)</code>	A is numeric or logical.	<code>isnumeric</code> , <code>islogical</code>
<code>mustBeReal(A)</code>	A has no imaginary part.	<code>isreal</code>

Name	Meaning	Functions Called on Inputs
mustBeInteger(A)	$A == \text{floor}(A)$	isreal, isfinite, floor, isnumeric, islogical

To specify a validation function when creating a setting, use the 'ValidationFcn' name-value pair argument and specify the function handle. For example, add the setting MyLogicalSetting to the mysettings settings group and specify that its value must be a logical scalar.

```
s = settings;
addGroup(s, 'mysettings');
addSetting(s.mysettings, 'MyLogicalSetting', 'ValidationFcn', @matlab.settings.mustBeLogicalScalar);
```

Try setting the value of MyLogicalSetting to a non-logical value. MATLAB returns an error.

```
s.mysettings.MyLogicalSetting.PersonalValue = 10;
```

```
Error setting 'MyLogicalSetting' in group 'mysettings': Value must be logical or convertible to logical.
```

You also can specify a validation function for an entire settings group. When specified, the validation function is used to validate the values of all settings within the group that do not have their own validation functions defined. For example, create the settings group mylogicalsettings and specify the validation function matlab.settings.mustBeLogicalScalar.

```
addGroup(s.mysettings, 'mylogicalsettings', 'ValidationFcn', @matlab.settings.mustBeLogicalScalar);
```

Create the setting MyLogicalSetting within the mylogicalsettings group and try setting the value of the setting to a non-logical value. MATLAB returns an error.

```
addSetting(s.mysettings.mylogicalsettings, 'MyLogicalSetting');
s.mysettings.mylogicalsettings.PersonalValue = 10;
```

```
Error setting 'MyLogicalSetting' in group 'mysettings': Value must be logical or convertible to logical.
```

Define Custom Validation Functions

You also can create your own validation functions. These can check for properties that are not covered by the MATLAB validation functions. Validation functions are ordinary MATLAB functions that are designed for the purpose of validating settings values. They must:

- Accept the potential setting value as an input argument.
- Have no output arguments.
- Throw an error if the validation fails.

Place validation functions on the MATLAB path to make them available.

For example, create a function to validate whether the value of a setting is numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
    assert(isnumeric(x),errorMsg);
end
```

Add the validation function to a new setting.

```
s = settings;
addGroup(s,'mysettings');
addSetting(s.mysettings,'MyNumericSetting','ValidationFcn',@numericValidationFcn);
```

Set the value of `MyNumericSetting` to a non-numeric value. MATLAB returns an error.

```
s.mysettings.MyNumericSetting.PersonalValue = 'Hello';
```

```
Unable to validate settings data. Error using myvalidationFcn (line 3)
Value must be numeric.
```

You also can create custom validation functions to make use of MATLAB validation functions that require multiple inputs, such as `mustBeGreaterThan`, `mustBeLessThan`, `mustBeGreaterThanOrEqual`, `mustBeLessThanOrEqual`, and `mustBeMember`. For example, this function validates that the value of a setting is one of four colors.

```
function colorValidationFcn(val)
    mustBeMember(val, ['Black' 'Blue' 'Yellow' 'Green']);
end
```

For more information about adding a validation function to a setting or settings group, see `addSetting` and `addGroup`.

See Also

`addGroup` | `addSetting` | `hasGroup` | `hasSetting` | `removeGroup` | `removeSetting` | `settings`

More About

- “Access and Modify Settings” on page 22-2

Create Factory Settings for Toolboxes

If you create a toolbox that works with MathWorks products, you can add settings to the toolbox that toolbox users can use to customize the appearance and behavior of the toolbox after installation. For example, you can add a setting that allows a user to change the font size in your toolbox.

To add settings that include factory values that ship with the toolbox, create factory settings. After installing your toolbox, users can then either use the factory values, or specify their own personal or temporary values.

Creating factory settings for toolboxes involves these steps:

- 1 Create the factory settings tree.
- 2 Create the factory settings JSON file.
- 3 Test the factory settings tree.

Create Factory Settings Tree

The first step to creating factory settings for a toolbox is to create the factory settings tree. Use the `matlab.settings.FactoryGroup.createToolboxGroup` function to create the root factory settings group for a toolbox. For example, create the root factory group for the toolbox `mytoolbox`. By default, factory groups are hidden, which means that they do not display in the parent settings group. Specify the `'Hidden'` name-value pair with a value of `false` to ensure that the group is visible in the factory settings tree.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', 'Hidden', false)
```

Once the toolbox root factory group is created, add factory settings and factory settings groups to the root to create the factory settings tree.

To add a new factory settings group, use the `addGroup` function. Specify the `'Hidden'` name-value pair with a value of `false` to create a visible factory group. For example, add the `font` factory group as a visible group.

```
toolboxFontGroup = addGroup(myToolboxFactoryTree, 'font', 'Hidden', false)
```

```
toolboxFontGroup =  
    FactoryGroup with properties:
```

```
    Name: "font"
```

```
ValidationFcn: []  
Hidden: 0
```

To add a new factory setting, use the `addSetting` function. For example, add `FontSize` as a visible factory setting in the font factory group. Specify a factory value for the setting. This value ships with the toolbox.

```
addSetting(toolboxFontGroup, 'FontSize', 'FactoryValue', 11, 'Hidden', false)
```

```
ans =
```

```
FactorySetting with properties:
```

```
        Name: "FontSize"  
    FactoryValue: 11  
FactoryValueFcn: []  
    ValidationFcn: []  
        Hidden: 0  
        ReadOnly: 0
```

You also can add read-only settings using the `'ReadOnly'` name-value pair argument. Add read-only settings to prevent toolbox users from changing the value of the settings by specifying a temporary or personal value for the setting.

Place all of the factory settings tree creation commands in a function with no inputs. Include the function with your toolbox code when you package and distribute the toolbox. For example, the function `createMyToolboxFactoryTree.mlx` creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()  
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...  
        'Hidden', false);  
  
    toolboxFontGroup = addGroup(myToolboxFactoryTree, 'font', 'Hidden', false)  
    addSetting(toolboxFontGroup, 'MyFontSize', 'FactoryValue', 11, 'Hidden', false, ...  
        'ValidationFcn', @matlab.settings.mustBeNumericScalar)  
    addSetting(toolboxFontGroup, 'MyFontColor', 'FactoryValue', 'Black', ...  
        'Hidden', false, 'ValidationFcn', @matlab.settings.mustBeStringScalar);  
end
```

Validate Settings Using Functions

You can impose specific restrictions on settings values by specifying a validation function for a setting or group. A validation function accepts a potential setting value as an argument, and throws an error if the value does not meet a specific requirement.

MATLAB defines several useful validation functions.

Name	Meaning	Functions Called on Inputs
<code>matlab.settings.mustBeStringScalar(A)</code>	A is a string array with one element.	<code>isStringScalar</code>
<code>matlab.settings.mustBeLogicalScalar(A)</code>	A is logical with one element.	<code>islogical</code> , <code>isscalar</code>
<code>matlab.settings.mustBeNumericScalar(A)</code>	A is numeric with one element.	<code>isnumeric</code> , <code>isscalar</code>
<code>matlab.settings.mustBeIntegerScalar(A)</code>	A is an integer with one element.	<code>isinteger</code> , <code>isscalar</code>
<code>mustBePositive(A)</code>	$A > 0$	<code>gt</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNonpositive(A)</code>	$A \leq 0$	<code>ge</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeFinite(A)</code>	A has no NaN and no Inf elements.	<code>isfinite</code>
<code>mustBeNonNaN(A)</code>	A has no NaN elements.	<code>isnan</code>
<code>mustBeNonnegative(A)</code>	$A \geq 0$	<code>ge</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNegative(A)</code>	$A < 0$	<code>lt</code> , <code>isreal</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNonzero(A)</code>	$A \neq 0$	<code>eq</code> , <code>isnumeric</code> , <code>islogical</code>
<code>mustBeNonempty(A)</code>	A is not empty.	<code>isempty</code>
<code>mustBeNonsparse(A)</code>	A has no sparse elements.	<code>issparse</code>
<code>mustBeNumeric(A)</code>	A is numeric.	<code>isnumeric</code>
<code>mustBeNumericOrLogical(A)</code>	A is numeric or logical.	<code>isnumeric</code> , <code>islogical</code>

Name	Meaning	Functions Called on Inputs
<code>mustBeReal(A)</code>	A has no imaginary part.	<code>isreal</code>
<code>mustBeInteger(A)</code>	<code>A == floor(A)</code>	<code>isreal</code> , <code>isfinite</code> , <code>floor</code> , <code>isnumeric</code> , <code>islogical</code>

To specify a validation function when creating a factory setting, use the 'ValidationFcn' name-value pair argument and specify the function handle. For example, add the setting `MyLogicalSetting` to the `myfactorysettings` group and specify that its value must be a logical scalar.

```
addSetting(s.myfactorysettings,'MyLogicalSetting','ValidationFcn', ...
    @matlab.settings.mustBeLogicalScalar);
```

Try setting the value of `MyLogicalSetting` to a nonlogical value. As expected, MATLAB throws an error.

```
s.myfactorysettings.MyLogicalSetting.PersonalValue = 10
```

```
Error setting "MyLogicalSetting" in group "myfactorysettings": Value must be logical or convertible to logical.
```

You also can specify a validation function for an entire factory settings group. When specified, the function is used to validate the values of all factory settings within the group that do not specify their own validation functions. This includes settings in subgroups, as long as the subgroup or settings do not specify their own validation functions. For example, create the settings group `mylogicalsettings` and specify the validation function `matlab.settings.mustBeLogicalScalar`.

```
addGroup(s.myfactorysettings,'mylogicalsettings','ValidationFcn', ...
    @matlab.settings.mustBeLogicalScalar);
```

Create the setting `MyLogicalSetting` within the `mylogicalsettings` group and try setting the value of the setting to a nonlogical value. As expected, MATLAB throws an error.

```
addSetting(s.myfactorysettings.mylogicalsettings,'MyLogicalSetting')
s.myfactorysettings.mylogicalsettings.PersonalValue = 10;
```

```
Error setting 'MyLogicalSetting' in group 'mysettings': Value must be logical or convertible to logical.
```

Define Custom Validation Functions

You also can create your own validation functions to check for factory settings properties that are not covered by the MATLAB validation functions. Validation functions are

ordinary MATLAB functions that are designed for the purpose of validating settings values. They must satisfy these conditions:

- Accept the potential setting value as an input argument.
- Have no output arguments.
- Throw an error if the validation fails.

Place validation functions on the MATLAB path to make them available.

For example, create a function to validate whether the value of a setting is numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
    assert(isnumeric(x),errorMsg);
end
```

Add this validation function to a new setting.

```
addSetting(s.mysettings, 'MyNumericSetting', 'ValidationFcn', @numericValidationFcn);
```

Set the value of `MyNumericSetting` to a nonnumeric value. As expected, MATLAB throws an error.

```
s.mysettings.MyNumericSetting.PersonalValue = 'Hello';
```

```
Unable to validate settings data. Error using myvalidationFcn (line 3)
Value must be numeric.
```

You also can create custom validation functions to make use of MATLAB validation functions that require multiple inputs, such as `mustBeGreaterThan`, `mustBeLessThan`, `mustBeGreaterThanOrEqual`, `mustBeLessThanOrEqual`, and `mustBeMember`. For example, this function validates that the value of a setting is one of four colors.

```
function colorValidationFcn(val)
    mustBeMember(val, ['Black' 'Blue' 'Yellow' 'Green']);
end
```

For more information about adding a validation function to a factory setting or factory settings group, see `addSetting` and `addGroup`.

Create Factory Settings JSON File

In order for MATLAB to know what function to use to create the factory settings tree, create a JSON-formatted file called `settingsInfo.json`. Include the file in the toolbox resources folder when you package and distribute the toolbox.

`settingsInfo.json` must follow this template. The `ToolboxGroupName` and `CreateTreeFcn` elements are required.

Note The values for `ToolboxGroupName` and `Hidden` must match the toolbox root factory group name and hidden state.

```
{
  "ToolboxGroupName" : "[toolbox root factory group name]",
  "Hidden" : "[hidden state of toolbox root factory group]",
  "CreateTreeFcn" : "[toolbox factory tree creation function]",
  "CreateUpgradersFcn" : "[toolbox factory tree upgrade function]"
}
```

For example, create the `settingsInfo.json` file for `mytoolbox`. Specify `mytoolbox` as the root settings group name, `false` as the hidden state, and `createMyToolboxFactoryTree` as the settings tree creation function.

```
{
  "ToolboxGroupName" : "mytoolbox",
  "Hidden" : "false",
  "CreateTreeFcn" : "createMyToolboxFactoryTree"
}
```

Test Factory Settings Tree

After creating the settings tree creation function and the `settingsInfo.json` file for your toolbox, you can test the settings tree before packaging and distributing the toolbox. Testing is useful for ensuring that the structure of the tree and the behavior of the settings are working as expected.

To test the tree, first, add the toolbox folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, use the `matlab.settings.reloadFactoryFile` function to load your toolbox factory settings and the `settings` function to access the root of the settings tree and your toolbox factory settings tree underneath it.

Note To avoid unexpected results, you must add the toolbox folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path.

For example, to test the factory settings tree for `mytoolbox`, you can run this code.

```
matlab.settings.reloadFactoryFile('mytoolbox');
s = settings;
s.mytoolbox.font.MyFontSize

ans =
    Setting 'mytoolbox.font.MyFontSize' with properties:
        ActiveValue: 11
        TemporaryValue: <no value>
        PersonalValue: <no value>
        FactoryValue: 11
```

Note

- The `matlab.settings.reloadFactoryFile` function is meant for debugging purposes only and should not be included in shipping toolbox code.
 - You must recreate any variables that reference the specified toolbox after calling `matlab.settings.reloadFactoryFile`. For example, if you create the variable `a = s.mytoolbox` and then call `matlab.settings.reloadFactoryFile`, you must recreate `a` to access the updated settings for `mytoolbox`.
-

Ensure Backward Compatibility Across Toolbox Versions

If you want to create a new version of your toolbox that includes modifications to the factory settings tree, you can take steps to ensure that any personal settings configured in a previously installed version of the toolbox are properly moved to the upgraded factory settings tree.

To ensure backward compatibility when making modifications to the factory settings tree, follow these steps:

- 1 Modify the factory settings tree.
- 2 Log changes to the tree.
- 3 Modify the factory settings JSON file.

4 Examine personal settings tree upgrade results.

Modifications that can cause backward incompatibility issues include renaming, moving, and removing toolbox factory settings or settings groups. If you are adding new settings to the factory settings tree, then you do not need to follow these steps.

Warning Changes to the factory settings tree creation function can affect the saved personal and temporary values of settings for the toolbox in question. To avoid data loss, back up your toolbox settings file before making any changes or before upgrading your toolbox. The toolbox settings file is located in your preferences folder, and has the name *toolboxname.mlsettings*. To see the full path for the preferences folder, type `prefdir` in the MATLAB Command Window.

If you experience unexpected changes to a toolbox settings tree after an upgrade, you can restore the tree by replacing the toolbox settings file with the backup that you created.

Update the Factory Settings Tree

The factory settings tree creation function creates the settings tree for the latest toolbox version. To update the factory settings tree for the latest toolbox version, modify the factory settings tree creation commands.

Note The commands in the factory settings tree creation function represents the latest toolbox version settings tree.

For example, suppose that in version 2 of `mytoolbox`, you want to rename the settings `MyFontSize` and `MyFontColor` to `FontSize` and `FontColor`. Change the settings names in `createMyToolboxFactoryTree.mlx`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden', false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree, 'font', 'Hidden', false)
    addSetting(toolboxFontGroup, 'FontSize', 'FactoryValue', 11, 'Hidden', false, ...
        'ValidationFcn', @matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup, 'FontColor', 'FactoryValue', 'Black', ...
        'Hidden', false, 'ValidationFcn', @matlab.settings.mustBeStringScalar);
end
```

Log Changes to the Tree

Create a function with no inputs to store the instructions for upgrading the personal settings from a previous version of the toolbox. Include the file with your toolbox code when you package and distribute the toolbox. Recording changes to the factory settings tree ensures that toolbox users upgrading to a new version do not have backward incompatibility issues with their existing toolbox settings. Modifications that can cause backward incompatibility issues include renaming, moving, and removing toolbox factory settings or settings group. You do not need to record the addition of new settings to the factory settings tree.

In the function, add a settings file upgrader object for each version of the toolbox that contains changes to the factory settings tree. Use the `move` and `remove` functions to record individual changes. For example, the function `createMyToolboxSettingsFileUpgraders.mlx` records the changes to `MyFontSize` and `MyFontColor` for version 2 of `mytoolbox`.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move("mytoolbox.font.MyFontSize","mytoolbox.font.FontSize");
    upgraders.move("mytoolbox.font.MyFontColor","mytoolbox.font.FontColor");
end
```

Note Do not modify recorded upgrade instructions in the settings tree upgrade function after packaging and distributing the toolbox to your users. Modifying instructions can have unexpected results. If you make additional changes to the settings tree, record the changes by appending them to the existing instructions, or by creating a new upgrader instance.

For example, this code records changes for version 2 and version 3 of `mytoolbox`.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move("mytoolbox.font.MyFontSize","mytoolbox.font.FontSize");
    upgraders.move("mytoolbox.font.MyFontColor","mytoolbox.font.FontColor");

    upgraders(2) = matlab.settings.SettingsFileUpgrader('Version3');
    upgraders(2).remove("mytoolbox.font.FontName");
end
```

Modify Factory Settings File

In order for MATLAB to know what function to use to upgrade the toolbox factory settings tree, specify the settings tree upgrade function in the factory settings file

(`settingsInfo.json`). For example, in `settingsInfo.json` for `mytoolbox`, specify `createMyToolboxSettingsFileUpgrader` as the settings tree upgrade function.

```
{
  "ToolboxGroupName" : "mytoolbox",
  "Hidden" : "false",
  "CreateTreeFcn" : "createMyToolboxFactoryTree",
  "CreateUpgradersFcn" : "createMyToolboxSettingsFileUpgraders"
}
```

Include the files `createMyToolboxFactoryTree.mlx`, `createMyToolboxSettingsFileUpgraders.mlx`, and `settingsInfo.json` when you package and distribute `mytoolbox`. Place the `settingsInfo.json` in the `toolbox` resources folder.

Note You must restart MATLAB after changing the `settingsInfo.json` file.

Examine Personal Settings Tree Upgrade Results

After upgrading the personal settings of a toolbox, you can examine the results to ensure that the upgrade occurred correctly. To examine the upgrade results, use the `matlab.settings.loadSettingsCompatibilityResults` function.

Examining the upgrade results before distributing a toolbox is useful to ensure that the upgrade will occur correctly after the toolbox has been distributed. For example, examine the upgrade results for version 2 of `mytoolbox` before distributing the toolbox.

- 1 Reload the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

- 2 Use the `settings` function to access the root of the settings tree and verify that the personal value for the `FontSize` setting was correctly moved over from the `MyFontSize` setting. Accessing your toolbox settings triggers the personal settings upgrade process.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans =
    Setting 'mytoolbox.font.FontSize' with properties:
        ActiveValue: 15
```

```

TemporaryValue: <no value>
PersonalValue: 15
FactoryValue: 11

```

- 3 Run the `matlab.settings.loadSettingsCompatibilityResults` function to get the upgrade results for version 2 of `mytoolbox`. Verify that there are no prevalidation exceptions.

```

matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2')

ans =
ReleaseCompatibilityResults with properties:
    VersionLabel: "Version2"
  PreValidationExceptions: [0x0 matlab.settings.ReleaseCompatibilityException]
                Results: [1x1 matlab.settings.VersionResults]

```

- 4 Access the `Results` property to determine the number of upgrade operations performed.

```

upgradeResults.Results

ans =
VersionResults with properties:
    VersionLabel: "Version2"
  VersionChanges: [1x2 matlab.settings.OperationResult]

```

- 5 Check the status of each upgrade operation to ensure that they were performed successfully.

```

upgradeResults.Results.VersionChanges.Status

ans =
"Succeeded"

ans =
"Succeeded"

```

You also can examine upgrade results after you have installed a new version of a toolbox. This approach is useful, for example, if you are helping a toolbox user troubleshoot their toolbox settings after an upgrade.

For example, suppose that you have version 1 of `mytoolbox` installed and have set personal values for several settings. After installing version 2 of `mytoolbox`, examine the upgrade results to ensure that your personal settings moved over correctly.

- 1 Use the `settings` function to access the root of the settings tree and your toolbox settings. Accessing your toolbox settings triggers the personal setting upgrade process.

```

s = settings;
s.mytoolbox

```

```
ans =  
  SettingsGroup 'mytoolbox' with properties:  
    font: [1x1 SettingsGroup]
```

- 2 Run the `matlab.settings.loadSettingsCompatibilityResults` function to get the upgrade results. Check the status of each upgrade operation performed to ensure that they were performed successfully.

```
upgradeResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox', 'Version2');  
upgradeResults.Results.VersionChanges.Status
```

```
ans =  
"Succeeded"
```

```
ans =  
"Succeeded"
```

Note

- After running the `matlab.settings.reloadFactoryFile` and `matlab.settings.loadSettingsCompatibilityResults` functions, delete the log of results before running the functions again. Deleting the log ensures the correct upgrade results are always loaded. The log is located in the preferences folder, in the `toolboxname` folder.
 - The `matlab.settings.reloadFactoryFile` and `matlab.settings.loadSettingsCompatibilityResults` functions are meant for debugging purposes only and should not be included in shipping toolbox code.
-

See Also

`matlab.settings.FactoryGroup.createToolboxGroup` |
`matlab.settings.loadSettingsCompatibilityResults` |
`matlab.settings.reloadFactoryFile`

More About

- “Access and Modify Settings” on page 22-2
- “Create Custom Settings” on page 22-6

Serial Port I/O

- “Serial Port Overview” on page 23-2
- “Create Serial Port Object” on page 23-16
- “Configure Serial Port Communication Settings” on page 23-19
- “Write and Read Serial Port Data” on page 23-21
- “Use Callbacks for Serial Port Communication” on page 23-26
- “Use Serial Port Control Pins” on page 23-28
- “Getting Started with Serial Port Communication” on page 23-34
- “Transition Your Code to serialport Interface” on page 23-40
- “Read Streaming Data from Arduino Using Serial Port Communication” on page 23-47
- “Capabilities and Supported Interfaces and Platforms” on page 23-50
- “Overview of the Serial Port” on page 23-53
- “Create a Serial Port Object” on page 23-67
- “Connect to the Device” on page 23-71
- “Configure Communication Settings” on page 23-72
- “Write and Read Data” on page 23-73
- “Events and Callbacks” on page 23-91
- “Control Pins” on page 23-99
- “Debugging: Recording Information to Disk” on page 23-105
- “Save and Load” on page 23-111
- “Disconnect and Clean Up” on page 23-113
- “Property Reference” on page 23-115
- “Properties — Alphabetical List” on page 23-118

Serial Port Overview

In this section...

“What Is Serial Communication?” on page 23-2

“Serial Port Interface Standard” on page 23-2

“Supported Platforms” on page 23-3

“Connecting Two Devices with a Serial Cable” on page 23-3

“Serial Port Signals and Pin Assignments” on page 23-4

“Serial Data Format” on page 23-8

“Find Serial Port Information for Your Platform” on page 23-11

What Is Serial Communication?

Serial communication is the most common low-level protocol for communicating between two or more devices. Normally, one device is a computer, while the other device can be a modem, a printer, another computer, or a scientific instrument such as an oscilloscope or a function generator.

As the name suggests, the serial port sends and receives bytes of information in a serial fashion—one bit at a time. These bytes are transmitted using either a binary format or a text (ASCII) format.

For many serial port applications, you can communicate with your instrument without detailed knowledge of how the serial port works. Communication is established through a serial port object, which you create in the MATLAB workspace.

If your application is straightforward, or if you are already familiar with the topics mentioned above, you might want to begin with “Create Serial Port Object” on page 23-16.

Serial Port Interface Standard

Over the years, several serial port interface standards for connecting computers to peripheral devices have been developed. These standards include RS-232, RS-422, and RS-485 — all of which are supported by the `serialport` object. The most widely used standard is RS-232.

The current version of this standard is designated TIA/EIA-232C, which is published by the Telecommunications Industry Association. However, the term “RS-232” is still in popular use, and is used here to refer to a serial communication port that follows the TIA/EIA-232 standard. RS-232 defines these serial port characteristics:

- Maximum bit transfer rate and cable length
- Names, electrical characteristics, and functions of signals
- Mechanical connections and pin assignments

Primary communication uses three pins: the Transmit Data pin, the Receive Data pin, and the Ground pin. Other pins are available for data flow control, but are not required.

Note This guide assumes that you are using the RS-232 standard. Refer to your device documentation to see which interface standard you can use.

Supported Platforms

The MATLAB serial port interface is supported on:

- Linux 64-bit
- macOS 64-bit
- Microsoft Windows 64-bit

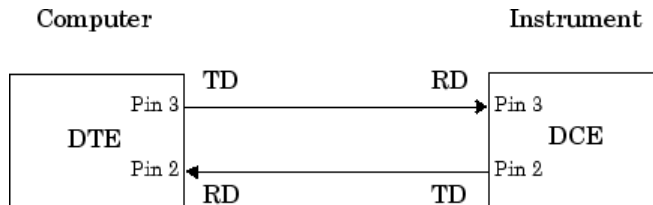
Connecting Two Devices with a Serial Cable

The RS-232 and RS-485 standard defines the two devices connected with a serial cable as the data terminal equipment (DTE) and data circuit-terminating equipment (DCE). This terminology reflects the RS-232 origin as a standard for communication between a computer terminal and a modem.

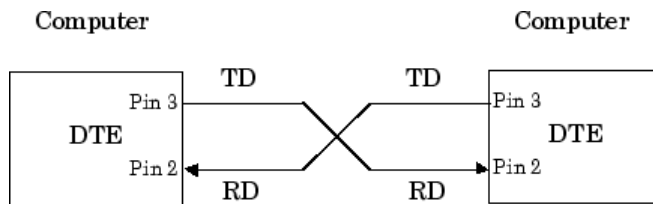
In this guide, your computer is considered a DTE, while peripheral devices such as modems and printers are considered DCEs. Note that many scientific instruments function as DTEs.

Because RS-232 mainly involves connecting a DTE to a DCE, the pin assignment definitions specify straight-through cabling, where pin 1 is connected to pin 1, pin 2 is connected to pin 2, and so on. A DTE-to-DCE serial connection using the transmit data

(TD) pin and the receive data (RD) pin is shown below. Refer to “Serial Port Signals and Pin Assignments” on page 23-4 for more information about serial port pins.



If you connect two DTEs or two DCEs using a straight serial cable, then the TD pin on each device is connected to the other, and the RD pin on each device is connected to the other. Therefore, to connect two like devices, you must use a *null modem* cable. As shown below, null modem cables cross the transmit and receive lines in the cable.

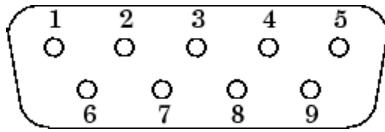


Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/RS-485 adaptor, then you can use the `serialport` object with these devices.

Serial Port Signals and Pin Assignments

Serial ports consist of two signal types: *data signals* and *control signals*. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most PCs and UNIX platforms use a 9-pin connection. In fact, only three pins are required for serial port communications: one for receiving data, one for transmitting data, and one for the signal ground.

The following figure shows a pin assignment scheme for a nine-pin male connector on a DTE.



This table describes the pins and signals associated with the nine-pin connector. Refer to the RS-232 or the RS-485 standard for a description of the signals and pin assignments for a 25-pin connector.

Serial Port Pin and Signal Assignments

Pin	Label	Signal Name	Signal Type
1	CD	Carrier Detect	Control
2	RD	Received Data	Data
3	TD	Transmitted Data	Data
4	DTR	Data Terminal Ready	Control
5	GND	Signal Ground	Ground
6	DSR	Data Set Ready	Control
7	RTS	Request to Send	Control
8	CTS	Clear to Send	Control
9	RI	Ring Indicator	Control

The term “data set” is synonymous with “modem” or “device,” while the term “data terminal” is synonymous with “computer.”

Note The serial port pin and signal assignments are with respect to the DTE. For example, data is transmitted from the TD pin of the DTE to the RD pin of the DCE.

Signal States

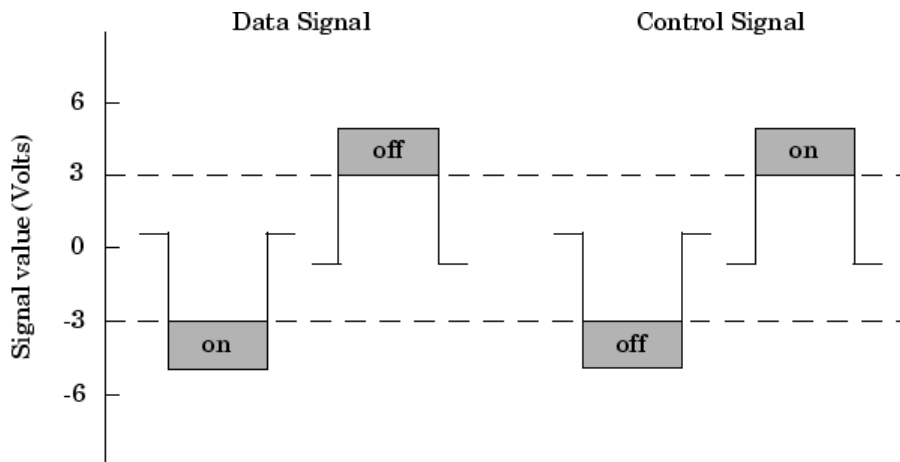
Signals can be in either an *active state* or an *inactive state*. An active state corresponds to the binary value 1, while an inactive state corresponds to the binary value 0. An active signal state is often described as *logic 1*, *on*, *true*, or a *mark*. An inactive signal state is often described as *logic 0*, *off*, *false*, or a *space*.

For data signals, the “on” state occurs when the received signal voltage is more negative than -3 volts, while the “off” state occurs for voltages more positive than 3 volts. For

control signals, the “on” state occurs when the received signal voltage is more positive than 3 volts, while the “off” state occurs for voltages more negative than -3 volts. The voltage between -3 volts and +3 volts is considered a transition region, and the signal state is undefined.

To bring the signal to the “on” state, the controlling device *unasserts* (or *lowers*) the value for data pins and *asserts* (or *raises*) the value for control pins. Conversely, to bring the signal to the “off” state, the controlling device asserts the value for data pins and unasserts the value for control pins.

The following figure depicts the “on” and “off” states for a data signal and for a control signal.



Data Pins

Most serial port devices support *full-duplex* communication, meaning that they can send and receive data at the same time. Therefore, separate pins are used for transmitting and receiving data. For these devices, the TD, RD, and GND pins are used. However, some types of serial port devices support only one-way or *half-duplex* communications. For these devices, only the TD and GND pins are used. This guide assumes that a full-duplex serial port is connected to your device.

The TD pin carries data transmitted by a DTE to a DCE. The RD pin carries data that is received by a DTE from a DCE.

Control Pins

The control pins of a nine-pin serial port are used to determine the presence of connected devices and control the flow of data. The control pins include:

- “RTS and CTS Pins” on page 23-7
- “DTR and DSR Pins” on page 23-7
- “CD and RI Pins” on page 23-8

RTS and CTS Pins

The RTS and CTS pins are used to signal whether the devices are ready to send or receive data. This type of data flow control — called hardware handshaking — is used to prevent data loss during transmission. When enabled for both the DTE and DCE, hardware handshaking using RTS and CTS follows these steps:

- 1** The DTE asserts the RTS pin to instruct the DCE that it is ready to receive data.
- 2** The DCE asserts the CTS pin, indicating that it is clear to send data over the TD pin. If data can no longer be sent, the CTS pin is unasserted.
- 3** The data is transmitted to the DTE over the TD pin. If data can no longer be accepted, the RTS pin is unasserted by the DTE and the data transmission is stopped.

To enable hardware handshaking, refer to “Controlling the Flow of Data: Handshaking” on page 23-31.

DTR and DSR Pins

Many devices use the DSR and DTR pins to signal if they are connected and powered. Signaling the presence of connected devices using DTR and DSR follows these steps:

- 1** The DTE asserts the DTR pin to request that the DCE connect to the communication line.
- 2** The DCE asserts the DSR pin to indicate that it is connected.
- 3** DCE unasserts the DSR pin when it is disconnected from the communication line.

The DTR and DSR pins were originally designed to provide an alternative method of hardware handshaking. However, the RTS and CTS pins are usually used in this way, and not the DSR and DTR pins. Refer to your device documentation to determine its specific pin behavior.

CD and RI Pins

The CD and RI pins are typically used to indicate the presence of certain signals during modem-modem connections.

CD is used by a modem to signal that it has made a connection with another modem, or has detected a carrier tone. CD is asserted when the DCE is receiving a signal of a suitable frequency. CD is unasserted if the DCE is not receiving a suitable signal.

RI is used to indicate the presence of an audible ringing signal. RI is asserted when the DCE is receiving a ringing signal. RI is unasserted when the DCE is not receiving a ringing signal (for example, it is between rings).

Serial Data Format

The serial data format includes one start bit, between five and eight data bits, and one stop bit. A parity bit and an additional stop bit might be included in the format as well. This diagram illustrates the serial data format.



The format for serial port data is often expressed using the following notation:

number of data bits - parity type - number of stop bits

For example, 8-N-1 is interpreted as eight data bits, no parity bit, and one stop bit, while 7-E-2 is interpreted as seven data bits, even parity, and two stop bits.

The data bits are often referred to as a *character* because these bits usually represent an ASCII character. The remaining bits are called *framing bits* because they frame the data bits.

Bytes Versus Values

The collection of bits that compose the serial data format is called a *byte*. At first, this term might seem inaccurate because a byte is 8 bits and the serial data format can range between 7 bits and 12 bits. However, when serial data is stored on your computer, the framing bits are stripped away, and only the data bits are retained. Moreover, eight data

bits are always used regardless of the number of data bits specified for transmission, with the unused bits assigned a value of 0.

When reading or writing data, you might need to specify a *value*, which can consist of one or more bytes. For example, if you read one value from a device using the `int32` format, then that value consists of four bytes. For more information about reading and writing values, refer to “Write and Read Serial Port Data” on page 23-21.

Synchronous and Asynchronous Communication

The RS-232 and the RS-485 standards support two types of communication protocols: synchronous and asynchronous.

Using the synchronous protocol, all transmitted bits are synchronized to a common clock signal. The two devices initially synchronize themselves to each other, and then continually send characters to stay synchronized. Even when actual data is not really being sent, a constant flow of bits allows each device to know where the other is at any given time. That is, each bit that is sent is either actual data or an idle character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

Using the asynchronous protocol, each device uses its own internal clock, resulting in bytes that are transferred at arbitrary times. So, instead of using time as a way to synchronize the bits, the data format is used.

In particular, the data transmission is synchronized using the start bit of the word, while one or more stop bits indicate the end of the word. The requirement to send these additional bits causes asynchronous communications to be slightly slower than synchronous. However, it has the advantage that the processor does not have to deal with the additional idle characters. Most serial ports operate asynchronously.

Note In this guide, the terms “synchronous” and “asynchronous” refer to whether read or write operations block access to the MATLAB Command Window.

How Are the Bits Transmitted?

By definition, serial data is transmitted one bit at a time. The order in which the bits are transmitted follows these steps:

- 1 The start bit is transmitted with a value of 0.
- 2 The data bits are transmitted. The first data bit corresponds to the least significant bit (LSB), while the last data bit corresponds to the most significant bit (MSB).
- 3 The parity bit (if defined) is transmitted.
- 4 One or two stop bits are transmitted, each with a value of 1.

The number of bits transferred per second is given by the *baud rate*. The transferred bits include the start bit, the data bits, the parity bit (if defined), and the stop bits.

Start and Stop Bits

As described in “Synchronous and Asynchronous Communication” on page 23-9, most serial ports operate asynchronously. This means that the transmitted byte must be identified by start and stop bits. The start bit indicates when the data byte is about to begin and the stop bit indicates when the data byte has been transferred. The process of identifying bytes with the serial data format follows these steps:

- 1 When a serial port pin is idle (not transmitting data), then it is in an “on” state.
- 2 When data is about to be transmitted, the serial port pin switches to an “off” state due to the start bit.
- 3 The serial port pin switches back to an “on” state due to the stop bit(s). This indicates the end of the byte.

Data Bits

The data bits transferred through a serial port can represent device commands, sensor readings, error messages, and so on. The data can be transferred as either binary data or as text (ASCII) data.

Most serial ports use between five and eight data bits. Binary data is typically transmitted as eight bits. Text-based data is transmitted as either seven bits or eight bits. If the data is based on the ASCII character set, then a minimum of seven bits is required because there are 2^7 or 128 distinct characters. If an eighth bit is used, it must have a value of 0. If the data is based on the extended ASCII character set, then eight bits must be used because there are 2^8 or 256 distinct characters.

Parity Bit

The parity bit provides simple error (parity) checking for the transmitted data. This table describes the types of parity checking.

Parity Types

Parity Type	Description
Even	The data bits plus the parity bit produce an even number of 1s.
Mark	The parity bit is always 1.
Odd	The data bits plus the parity bit produce an odd number of 1s.
Space	The parity bit is always 0.

Mark and space parity checking are seldom used because they offer minimal error detection. You can choose not to use parity checking at all.

The parity checking process follows these steps:

- 1 The transmitting device sets the parity bit to 0 or to 1 depending on the data bit values and the type of parity checking selected.
- 2 The receiving device checks if the parity bit is consistent with the transmitted data. If it is, then the data bits are accepted. If it is not, then an error is returned.

Note Parity checking can detect only one-bit errors. Multiple-bit errors can appear as valid data.

For example, suppose the data bits 01110001 are transmitted to your computer. If even parity is selected, then the parity bit is set to 0 by the transmitting device to produce an even number of 1s. If odd parity is selected, then the parity bit is set to 1 by the transmitting device to produce an odd number of 1s.

Find Serial Port Information for Your Platform

You can find serial port information using the resources provided by Windows and UNIX platforms.

Note Your operating system provides default values for all serial port settings. However, these settings are overridden by your MATLAB code, and have no effect on your serial port application.

Use the serialportlist Function to Find Available Ports

The `serialportlist` function returns a list of all serial ports on a system, including virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. The function provides a list of the serial ports that you have access to on your computer and can use for serial port communication. For example:

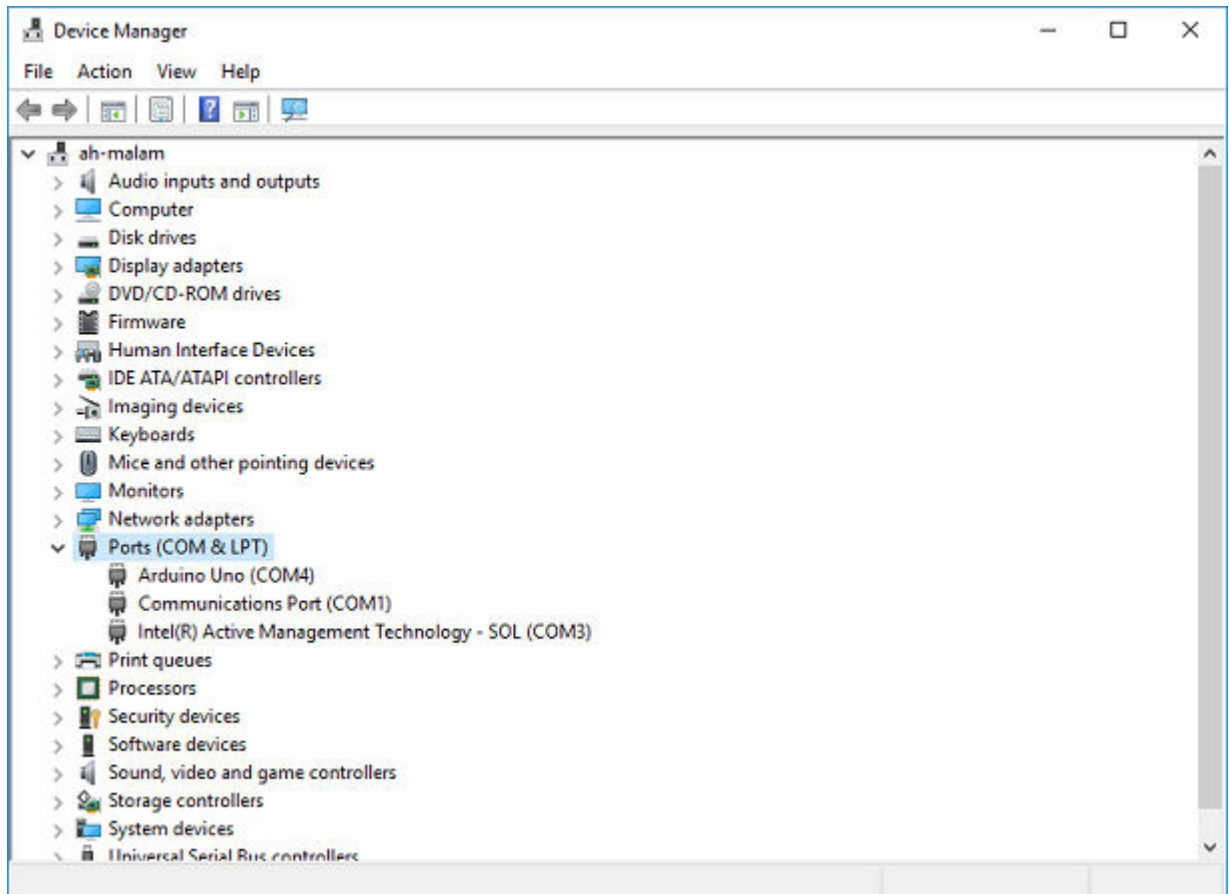
```
serialportlist  
  
ans =  
  
    1x3 string array  
  
    "COM1"    "COM3"    "COM4"
```

Note The `serialportlist` function shows both available and in-use ports on Windows and macOS systems, but on Linux, it shows only available ports and not in-use ports.

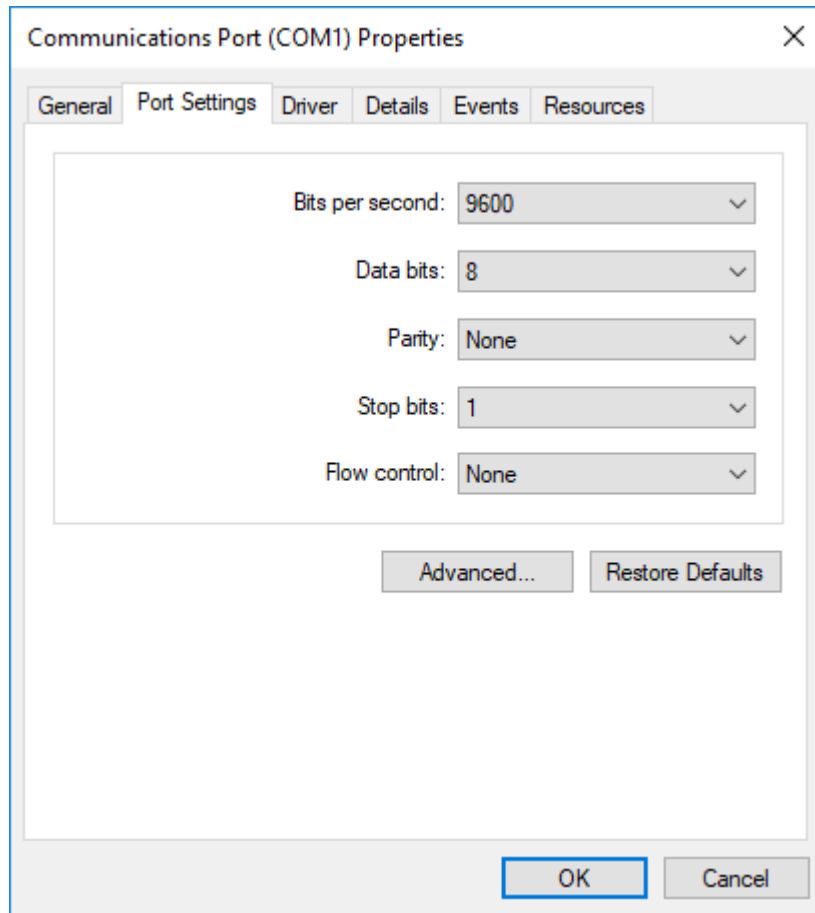
Windows Platform

You can access serial port information through the **Device Manager**.

- 1 Open **Device Manager**.
- 2 Expand the **Ports (COM & LPT)** list.



- 3 Double-click the **Communications Port (COM1)** item.
- 4 Select the **Port Settings** tab.



UNIX Platform

To find serial port information for UNIX platforms, you need to know the serial port names. These names can vary between different operating systems.

On Linux, serial port devices are typically named `ttyS0`, `ttyS1`, and so on. You can use the `setserial` command to display or configure serial port information. For example, to display which serial ports are available:

```
setserial -bg /dev/ttyS*
```

```
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A  
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

To display detailed information about `ttyS0`:

```
setserial -ag /dev/ttyS0
```

```
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4  
  Baud_base: 115200, close_delay: 50, divisor: 0  
  closing_wait: 3000, closing_wait2: infinte  
  Flags: spd_normal skip_test session_lockout
```

Note If the `setserial -ag` command does not work, make sure that you have read and write permission for the port.

For all supported UNIX platforms, including macOS, you can use the `stty` command to display or configure serial port information. For example, to display serial port properties for `ttyS0`, type:

```
stty -a < /dev/ttyS0
```

To configure the baud rate as 4800 bits per second, type:

```
stty speed 4800 < /dev/ttyS0 > /dev/ttyS0
```

Note This example shows how to set `tty` parameters, not the baud rate. To set the baud rate using the MATLAB serial interface, refer to “Configure Serial Port Communication Settings” on page 23-19.

Create Serial Port Object

In this section...

“Create a Serial Port Object” on page 23-16

“Serial Port Object Display” on page 23-17

Create a Serial Port Object

You create a serial port object with the `serialport` function. `serialport` requires the name of the serial port connected to your device and the baud rate as input arguments. You can also configure property values during object creation using name-value pair arguments.

Each serial port object is associated with one serial port. For example, connect to a device that is on serial port COM1 and configured for a baud rate of 4800.

```
s = serialport("COM1",4800);
```

If the specified port does not exist, or if it is in use, you cannot connect the serial port object to the device. The port name depends on the platform that the serial port is on.

You can also use the `serialportlist` function to return a list of all serial ports on a system, including virtual serial ports provided by USB-to-serial devices and Bluetooth® Serial Port Profile devices. The list shows all serial ports that you have access to on your computer and can use for serial port communication.

```
serialportlist
ans =
    1x3 string array
    "COM1"    "COM3"    "COM4"
```

This table shows an example of serial constructors on different platforms.

Platform	Serial Constructor
Linux 64-bit	<code>s = serialport("/dev/ttyS0",9600);</code>
macOS 64-bit	<code>s = serialport("/dev/tty.KeySerial1",9600);</code>

Platform	Serial Constructor
Microsoft Windows 64-bit	<code>s = serialport("COM1",9600);</code>

Note The first time you try to access a serial port in MATLAB using the `s = serialport("COM1",9600)` call, make sure that the port is free and is not already open in any other application. If the port is open in another application, MATLAB cannot access it. After you access the serial port in MATLAB, you can open the same port in other applications, and MATLAB continues to use it along with any other application that has it open as well.

Serial Port Object Display

The serial port object provides a convenient display that summarizes important configuration and state information. You can invoke the display summary in three ways:

- Type the serial port object variable name at the command line.
- Exclude the semicolon when creating a serial port object.
- Exclude the semicolon when configuring properties using dot notation.

You can also display summary information using the workspace browser by right-clicking an instrument object and selecting **Display Summary** from the context menu.

The display summary for the serial port object `s` on a Windows machine is given here.

```
s = serialport("COM4",9600)
```

```
s =
```

```
Serialport with properties
```

```

    Port: "COM4"
    BaudRate: 9600
    NumBytesAvailable: 0
    NumBytesWritten: 0
```

```
Show all properties
```

```

    Port: "COM4"
    BaudRate: 9600
    NumBytesAvailable: 0
```

```
NumBytesWritten: 0
    ByteOrder: "little-endian"
    DataBits: 8
    StopBits: 1
    Parity: "none"
    FlowControl: "none"
    Timeout: 10
    Terminator: "LF"

    BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
    BytesAvailableFcn: []

    ErrorOccurredFcn: []
    UserData: []
```

Use dot notation to configure and display property values.

```
s.BaudRate = 4800;
s.BaudRate
```

```
ans =
```

```
4800
```


Configure Serial Port Communication Settings

Before you can write or read data, both the serial port object and the instrument must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the “Serial Data Format” on page 23-8. These properties are as follows.

Serial Port Communication Properties

Property Name	Description
BaudRate	Specify the rate at which bits are transmitted.
Parity	Specify the type of parity checking.
DataBits	Specify the number of data bits to transmit.
StopBits	Specify the number of bits used to indicate the end of a byte.
Terminator	Specify the terminator character.

Caution If the serial port object and the instrument communication settings are not identical, you might not be able to successfully read or write data.

Refer to your instrument documentation for an explanation of its supported communication settings.

You can display the communication property values for the serial port object `s` created in “Create Serial Port Object” on page 23-16.

```
s = serialport("COM4", 9600)
```

```
s =
```

```
Serialport with properties
```

```

        Port: "COM4"
        BaudRate: 9600
    NumBytesAvailable: 0
    NumBytesWritten: 0

```

```
Show all properties
```

```
Port: "COM4"
```

```
        BaudRate: 9600
NumBytesAvailable: 0
        NumBytesWritten: 0

        ByteOrder: "little-endian"
        DataBits: 8
        StopBits: 1
        Parity: "none"
FlowControl: "none"
        Timeout: 10
        Terminator: "LF"

BytesAvailableFcnMode: "off"
BytesAvailableFcnCount: 64
BytesAvailableFcn: []

ErrorOccurredFcn: []
        UserData: []
```

Write and Read Serial Port Data

In this section...

“Rules for Completing Write and Read Operations” on page 23-21

“Writing and Reading Text Data” on page 23-21

“Writing and Reading Binary Data” on page 23-23

Rules for Completing Write and Read Operations

Completing Write Operations

A write operation using `write` or `writeline` completes when one of these conditions is satisfied:

- The specified data is written.
- The time specified by the `Timeout` property passes.

A text command is processed by the instrument only when it receives the required terminator. For serial port objects, each occurrence of `\n` in the ASCII command is replaced with the `Terminator` property value. The default value of `Terminator` is the line feed character. Refer to the documentation for your instrument to determine the terminator required by your instrument.

Completing Read Operations

A read operation with `read` or `readline` completes when one of these conditions is satisfied:

- The specified number of values is read.
- The time specified by the `Timeout` property passes.
- The terminator specified by the `Terminator` property is read.

Writing and Reading Text Data

This example illustrates how to communicate with a serial port instrument by writing and reading text data.

The instrument is a Tektronix® TDS 210 two-channel oscilloscope connected to the serial port COM1. Therefore, many of the commands in the example are specific to this

instrument. A sine wave is input into channel 2 of the oscilloscope, and you want to measure the peak-to-peak voltage of the input signal.

These functions and properties are used when reading and writing text.

Function	Purpose
<code>readline</code>	Read text data from the instrument.
<code>writeline</code>	Write text data to the instrument.
"Terminator"	Character used to terminate commands sent to the instrument.

Note This example is Windows specific.

- 1 Create a serial port object** — Create the serial port object `s` associated with the serial port COM1.

```
s = serialport("COM1",9600);
```

- 2 Write and read data** — Write the `*IDN?` command to the instrument using `writeline`, and then read back the result of the command using `readline`.

```
writeline(s,"*IDN?")  
s.NumBytesAvailable
```

```
ans =
```

```
    56
```

```
idn = readline(s)
```

```
idn =
```

```
"TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"
```

You need to determine the measurement source. Possible measurement sources include channel 1 and channel 2 of the oscilloscope.

```
writeline(s,"MEASUREMENT:IMMED:SOURCE?")  
source = readline(s)
```

```
source =
```

```
"CH1"
```

The scope is configured to return a measurement from channel 1. Because the input signal is connected to channel 2, you must configure the instrument to return a measurement from this channel.

```
writeline(s, "MEASUREMENT:IMMED:SOURCE CH2")
writeline(s, "MEASUREMENT:IMMED:SOURCE?")
source = readline(s)
```

```
source =
    "CH2"
```

You can now configure the scope to return the peak-to-peak voltage, and then request the value of this measurement.

```
writeline(s, "MEASUREMENT:MEAS1:TYPE PK2PK")
writeline(s, "MEASUREMENT:MEAS1:VALUE?")
```

Read back the result using the `readline` function.

```
ptop = readline(s)

ptop =
    "2.0199999809E0"
```

- 3 Disconnect and clean up** — Clear the serial port object `s` from the MATLAB workspace when you are done working with it.

```
clear s
```

Writing and Reading Binary Data

This example explores binary read and write operations with a serial port object. The instrument used is a Tektronix® TDS 210 oscilloscope.

Functions and Properties

These functions are used when reading and writing binary data.

Function	Purpose
<code>read</code>	Read binary data from the instrument.
<code>write</code>	Write binary data to the instrument.

Configure and Connect to the Serial Object

You need to create a serial object. In this example, create a serial port object associated with the COM1 port.

```
s = serialport("COM1",9600);
```

Write Binary Data

You use the `write` function to write binary data to the instrument. A binary write operation completes when one of these conditions is satisfied:

- All the data is written.
- A timeout occurs as specified by the `Timeout` property.

Note When you perform a write operation, think of the transmitted data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Writing Int16 Binary Data

Write a waveform as an `int16` array.

```
write(s,"Data:Destination RefB","string");  
write(s,"Data:Encdg SRPbinary","string");  
write(s,"Data:Width 2","string");  
write(s,"Data:Start 1","string");  
  
t = (0:499) .* 8 * pi / 500;  
data = round(sin(t) * 90 + 127);  
write(s,"CURVE #3500","string");
```

Note that one `int16` value consists of two bytes. Therefore, the following command writes 1000 bytes.

```
write(s,data,"int16")
```

Reading Binary Data

You use the `read` function to read binary data from the instrument. A binary read operation completes when one of these conditions is satisfied:

- A timeout occurs as specified by the `Timeout` property.
- The specified number of values is read.

Note When you perform a read operation, think of the received data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Reading `int16` Binary Data

Read the same waveform on channel 1 as an `int16` array.

```
read(s, "Data:Source CH1", "string");  
read(s, "Data:Encdg SRPbinary", "string");  
read(s, "Data:Width 2", "string");  
read(s, "Data:Start 1", "string");  
read(s, "Data:Stop 2500", "string");  
read(s, "Curve?", "string")
```

Note that one `int16` value consists of two bytes. Therefore, the following command reads 512 bytes.

```
data = read(s, 256, "int16");
```

Disconnect and Clean Up

If you are finished with the serial port object, clear the object from the workspace.

```
clear s
```

Use Callbacks for Serial Port Communication

In this section...

“Callback Properties” on page 23-26

“Using Callbacks” on page 23-26

Callback Properties

The properties and functions associated with callbacks are as follows.

Property or Function	Purpose
NumBytesAvailable	Number of bytes available to read
BytesAvailableFcn	Bytes available callback function
BytesAvailableFcnCount	Number of bytes of data to trigger callback
BytesAvailableFcnMode	Bytes available callback trigger mode
configureCallback	Set serial port callback function and trigger

Using Callbacks

This example uses a loopback device with the callback function `readSerialData` to return data to the command line when a terminator is read.

Note This example is Windows specific.

- 1 Create the callback function** — Define a callback function `readSerialData` that performs a terminated string read and returns the data.

```
function readSerialData(src,~)
    data = readline(src);
    disp(data);
end
```

- 2 Create an instrument object** — Create the serial port object `s` associated with serial port COM1.

```
s = serialport("COM1",9600);
```


- 3 Configure properties** — Configure `s` to execute the callback function `readSerialData` when the terminator is read.

```
configureCallback(s, "terminator", @readSerialData)
```

- 4 Disconnect and clean up** — Clear the objects from the MATLAB workspace when you are done.

```
clear s
```

Use Serial Port Control Pins

In this section...

“Control Pins” on page 23-28

“Signaling the Presence of Connected Devices” on page 23-28

“Controlling the Flow of Data: Handshaking” on page 23-31

Control Pins

As described in “Serial Port Signals and Pin Assignments” on page 23-4, nine-pin serial ports include six control pins. The functions and properties associated with the serial port control pins are as follows.

Function	Purpose
<code>getpinstatus</code>	Get serial pin status.
<code>setRTS</code>	Specify the state of the RTS pin.
<code>setDTR</code>	Specify the state of the DTR pin.
<code>FlowControl</code>	Specify the data flow control method to use.

Signaling the Presence of Connected Devices

DTEs and DCEs often use the CD, DSR, RI, and DTR pins to indicate whether a connection is established between serial port devices. Once the connection is established, you can begin to write or read data.

You can monitor the state of the CD, DSR, and RI pins with the `getpinstatus` function. You can specify the state of the DTR pin with the `setDTR` function.

The following example illustrates how these pins are used when two modems are connected to each other.

Connect Two Modems

This example (shown on a Windows machine) connects two modems to each other through the same computer, and illustrates how you can monitor the communication status for the computer-modem connections, and for the modem-modem connection. The first modem is connected to COM1, while the second modem is connected to COM2.

- 1 Connect to the instruments** — After the modems are powered on, the serial port object `s1` is created for the first modem, and the serial port object `s2` is created for the second modem. Both modems are configured for a baud rate of 9600 bits per second.

```
s1 = serialport("COM1",9600);  
s2 = serialport("COM2",9600);
```

You can verify that the modems (data sets) are ready to communicate with the computer by examining the value of the Data Set Ready pin using the `getpinstatus` function.

```
getpinstatus(s)
```

```
ans =
```

```
struct with fields:
```

```
    ClearToSend: 1  
    DataSetReady: 1  
    CarrierDetect: 0  
    RingIndicator: 0
```

The value of the `DataSetReady` field is 1, or `true`, because both modems were powered on before they were connected to the objects.

- 2 Configure properties** — Both modems are configured for a carriage return (CR) terminator using the `configureTerminator` function.

```
configureTerminator(s1,"CR")  
configureTerminator(s2,"CR")
```

- 3 Write and read data** — Write the `atd` command to the first modem using the `writeline` function. This command puts the modem “off the hook,” and is equivalent to manually lifting a phone receiver.

```
writeline(s1,'atd')
```

Write the `ata` command to the second modem using the `writeline` function. This command puts the modem in “answer mode,” which forces it to connect to the first modem.

```
writeline(s2,'ata')
```

After the two modems negotiate their connection, you can verify the connection status by examining the value of the Carrier Detect pin using the `getpinstatus` function.

```
getpinstatus(s)
```

```
ans =
```

```
    struct with fields:
```

```
        ClearToSend: 1
        DataSetReady: 1
        CarrierDetect: 1
        RingIndicator: 0
```

You can also verify the modem-modem connection by reading the descriptive message returned by the second modem.

```
s2.NumBytesAvailable
```

```
ans =
```

```
    25
```

```
out = read(s2,25,"uint32")
```

```
out =
```

```
ata
```

```
CONNECT 2400/NONE
```

Now break the connection between the two modems by using the `setDTR` function. You can verify that the modems are disconnected by examining the Carrier Detect pin value using the `getpinstatus` function.

```
setDTR(s1,false)
```

```
getpinstatus(s1)
```

```
ans =
```

```
    struct with fields:
```

```
        ClearToSend: 1
        DataSetReady: 1
        CarrierDetect: 0
        RingIndicator: 0
```

- 4 Disconnect and clean up** — Clear the objects from the MATLAB workspace when you are done.

```
clear s1 s2
```

Controlling the Flow of Data: Handshaking

Data flow control or *handshaking* is a method used for communicating between a DCE and a DTE to prevent data loss during transmission. For example, suppose your computer can receive only a limited amount of data before it must be processed. As this limit is reached, a handshaking signal is transmitted to the DCE to stop sending data. When the computer can accept more data, another handshaking signal is transmitted to the DCE to resume sending data.

If supported by your device, you can control data flow using one of these methods:

- “Hardware Handshaking” on page 23-31
- “Software Handshaking” on page 23-32

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

You can specify the data flow control method with the `FlowControl` property. If `FlowControl` is `hardware`, then hardware handshaking is used to control data flow. If `FlowControl` is `software`, then software handshaking is used to control data flow. If `FlowControl` is `none`, then no handshaking is used.

Hardware Handshaking

Hardware handshaking uses specific serial port pins to control data flow. In most cases, these are the RTS and CTS pins. Hardware handshaking using these pins is described in “RTS and CTS Pins” on page 23-7.

If `FlowControl` is `hardware`, then the RTS and CTS pins are automatically managed by the DTE and DCE. You can return the CTS pin value with the `getpinstatus` function. You can configure the RTS pin value with the `setRTS` function.

Note Some devices also use the DTR and DSR pins for handshaking. However, these pins are typically used to indicate that the system is ready for communication, and are not used to control data transmission. In MATLAB, hardware handshaking always uses the RTS and CTS pins.

If your device does not use hardware handshaking in the standard way, then you might need to manually configure the RTS pin using the `setRTS` function. In this case, configure `FlowControl` to `none`. If `FlowControl` is hardware, then the RTS value that you specify might not be honored. Refer to the device documentation to determine its specific pin behavior.

Software Handshaking

Software handshaking uses specific ASCII characters to control data flow. The following table describes these characters, known as Xon and Xoff (or XON and XOFF).

Software Handshaking Characters

Character	Integer Value	Description
Xon	17	Resume data transmission.
Xoff	19	Pause data transmission.

When you use software handshaking, the control characters are sent over the transmission line the same way as regular data. Therefore, you need only the TD, RD, and GND pins.

The main disadvantage of software handshaking is that you cannot write the Xon or Xoff characters while numerical data is being written to the instrument. This is because numerical data might contain a 17 or 19, which makes it impossible to distinguish between the control characters and the data. However, you can write Xon or Xoff while data is being asynchronously read from the instrument because you are using both the TD and RD pins.

Using Software Handshaking

Suppose you want to use software flow control in conjunction with your serial port application. To do this, you must configure the instrument and the serial port object for software flow control. For a serial port object `s` connected to a Tektronix TDS 210 oscilloscope, this configuration is accomplished with the following commands.

```
writeline(s,"RS232:S0FTF 0N")  
s.FlowControl = "software";
```

To pause data transfer, you write the numerical value 19 (Xoff) to the instrument.

```
write(s,19,"uint32");
```

To resume data transfer, you write the numerical value 17 (Xon) to the instrument.

```
write(s,17,"uint32");
```

Getting Started with Serial Port Communication

In this section...
"Query a Serial Port Device" on page 23-34
"The Serial Port Session" on page 23-34
"Configuring and Returning Properties" on page 23-35

Query a Serial Port Device

This example illustrates some basic serial port commands.

Note This example is shown on a Windows platform.

If you have a device connected to the serial port COM1 and configured for a baud rate of 4800, execute the following example.

```
s = serialport("COM1",4800);  
writeline(s,"*IDN?")  
out = readline(s);  
clear s
```

The `*IDN?` command queries the device for identification information, which is returned to `out`. If your device does not support this command, or if it is connected to a different serial port, modify the previous example accordingly.

Note `*IDN?` is one of the commands supported by the Standard Commands for Programmable Instruments (SCPI) language, which is used by many modern devices. Refer to your device documentation to see if it supports the SCPI language.

The Serial Port Session

This example describes the steps you use to perform any serial port task from beginning to end.

The serial port session comprises all the steps you are likely to take when communicating with a device connected to a serial port. These steps are:

- 1 Find your serial ports — Display a list of serial ports on your system using the `serialportlist` function.
- 2 Connect to a serial port device — Connect to a device for a specific serial port using the `serialport` creation function.

Configure properties during object creation if necessary. In particular, you might want to configure properties associated with serial port communications such as the baud rate, the number of data bits, and so on. Alter the necessary device settings by configuring property values, read data, and write data.

- 3 Configure properties — To establish the desired serial port object behavior, assign values to properties using dot notation.

In practice, you can configure many of the properties at any time including during, or just after, object creation. Conversely, depending on your device settings and the requirements of your serial port application, you might be able to accept the default property values and skip this step.

- 4 Write and read data — Write data to the device using the `writeline` or `write` function, and read data from the device using the `readline` or `read` function.

The serial port object behaves according to the previously configured or default property values.

- 5 Disconnect and clean up — When you no longer need the serial port object, remove it from the MATLAB workspace using the `clear` command.

The serial port session is reinforced in many of the serial port documentation examples. To see a basic example that uses the steps shown above, see “Query a Serial Port Device” on page 23-34.

Configuring and Returning Properties

This example describes how you display serial port property names and property values, and how you assign values to properties.

You establish the desired serial port object behavior by configuring property values. You can display or configure property values using the `set` function, the `get` function, or dot notation.

Displaying Property Names and Property Values

After you create the serial port object, use the `set` function to display all the configurable properties to the command line. Additionally, if a property has a finite set of string values, `set` also displays these values.

```
s = serial('COM1');
set(s)
    ByteOrder: [ {littleEndian} | bigEndian ]
    BytesAvailableFcn
    BytesAvailableFcnCount
    BytesAvailableFcnMode: [ {terminator} | byte ]
    ErrorFcn
    InputBufferSize
    Name
    OutputBufferSize
    OutputEmptyFcn
    RecordDetail: [ {compact} | verbose ]
    RecordMode: [ {overwrite} | append | index ]
    RecordName
    Tag
    Timeout
    TimerFcn
    TimerPeriod
    UserData

SERIAL specific properties:
    BaudRate
    BreakInterruptFcn
    DataBits
    DataTerminalReady: [ {on} | off ]
    FlowControl: [ {none} | hardware | software ]
    Parity: [ {none} | odd | even | mark | space ]
    PinStatusFcn
    Port
    ReadAsyncMode: [ {continuous} | manual ]
    RequestToSend: [ {on} | off ]
    StopBits
    Terminator
```

Use the `get` function to display one or more properties and their current values to the command line. To display all properties and their current values:

```
get(s)
    ByteOrder = littleEndian
```

```
BytesAvailable = 0
BytesAvailableFcn =
BytesAvailableFcnCount = 48
BytesAvailableFcnMode = terminator
BytesToOutput = 0
ErrorFcn =
InputBufferSize = 512
Name = Serial-COM1
OutputBufferSize = 512
OutputEmptyFcn =
RecordDetail = compact
RecordMode = overwrite
RecordName = record.txt
RecordStatus = off
Status = closed
Tag =
Timeout = 10
TimerFcn =
TimerPeriod = 1
TransferStatus = idle
Type = serial
UserData = []
ValuesReceived = 0
ValuesSent = 0
```

SERIAL specific properties:

```
BaudRate = 9600
BreakInterruptFcn =
DataBits = 8
DataTerminalReady = on
FlowControl = none
Parity = none
PinStatus = [1x1 struct]
PinStatusFcn =
Port = COM1
ReadAsyncMode = continuous
RequestToSend = on
StopBits = 1
Terminator = LF
```

To display the current value for one property, supply the property name to `get`.

```
get(s, 'OutputBufferSize')
ans =
    512
```

To display the current values for multiple properties, include the property names as elements of a cell array.

```
get(s,{'Parity','TransferStatus'})  
ans =  
    'none'    'idle'
```

Use the dot notation to display a single property value.

```
s.Parity  
ans =  
none
```

Configuring Property Values

You can configure property values using the `set` function:

```
set(s,'BaudRate',4800);
```

or the dot notation:

```
s.BaudRate = 4800;
```

To configure values for multiple properties, supply multiple property name/property value pairs to `set`.

```
set(s,'DataBits',7,'Name','Test1-serial')
```

Note that you can configure only one property value at a time using the dot notation.

In practice, you can configure many of the properties at any time while the serial port object exists — including during object creation. However, some properties are not configurable while the object is connected to the device or when recording information to disk. For information about when a property is configurable, see “Property Reference” on page 23-115.

Specifying Property Names

Serial port property names are presented using mixed case. While this makes property names easier to read, use any case you want when specifying property names. For example, to configure the `BaudRate` property:

```
s.BaudRate = 4800;  
s.baudrate = 4800;
```

Default Property Values

Whenever you do not explicitly define a value for a property, the default value is used. All configurable properties have default values.

Note Your operating system provides default values for all serial port settings such as the baud rate. However, these settings are overridden by your MATLAB code and have no effect on your serial port application.

If a property has a finite set of string values, the default value is enclosed by `{}`. For example, the default value for the `Parity` property is `none`.

```
set(s, 'Parity')  
[ {none} | odd | even | mark | space ]
```

You can find the default value for any property in the property reference pages.

Transition Your Code to serialport Interface

The `serial` function, its object functions, and its properties are not recommended. Use `serialport` instead.

serial Interface	serialport Interface	Example
<code>seriallist</code>	<code>serialportlist</code>	“Discover Serial Port Devices” on page 23-40
<code>serial</code>	<code>serialport</code>	“Connect to Serial Port Device” on page 23-41
<code>fwrite</code> and <code>fread</code>	<code>write</code> and <code>read</code>	“Read and Write” on page 23-41
<code>fprintf</code>	<code>writeline</code>	“Send a Command” on page 23-42
<code>fscanf</code> , <code>fgetl</code> , and <code>fgets</code>	<code>readline</code>	“Read a Terminated String” on page 23-43
<code>flushinput</code> and <code>flushoutput</code>	<code>flush</code>	“Flush Data from Memory” on page 23-44
<code>Terminator</code>	<code>configureTerminator</code>	“Set Terminator” on page 23-44
<code>BytesAvailableFcnCount</code> , <code>BytesAvailableFcnMode</code> , <code>BytesAvailableFcn</code> , and <code>BytesAvailable</code>	<code>configureCallback</code>	“Set Up a Callback Function” on page 23-45
<code>PinStatus</code>	<code>getpinstatus</code>	“Read Serial Pin Status” on page 23-46
<code>DataTerminalReady</code> and <code>RequestToSend</code>	<code>setDTR</code> and <code>setRTS</code>	“Set Serial DTR and RTS Pin States” on page 23-46

Discover Serial Port Devices

`seriallist` is not recommended. Use `serialportlist` instead.

Connect to Serial Port Device

This example shows how to connect to a serial port device using the recommended functionality.

Functionality	Use This Instead
<pre>s = serial("COM1"); s.BaudRate = 115200; fopen(s)</pre>	<pre>s = serialport("COM1",115200);</pre>

For more information, see `serialport`.

Read and Write

These examples use a loopback device to show how to perform a binary write and read, write a nonterminated command string, and read a fixed-length response string using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object fwrite(s,1:5,"uint32") data = fread(s,5,"uint32") data = 1 2 3 4 5</pre>	<pre>% s is a serialport object write(s,1:5,"uint32") data = read(s,5,"uint32") data = 1 2 3 4 5</pre>
<pre>% s is a serial object command = "start"; fwrite(s,command,"char")</pre>	<pre>% s is a serialport object command = "start"; write(s,command,"char")</pre>
	<pre>% s is a serialport object command = "start"; write(s,command,"string")</pre>

Functionality	Use This Instead
<pre>% s is a serial object length = 5; resp = fread(s,length,"char") resp = 115 116 97 114 116 resp = char(resp) ' resp = 'start'</pre>	<pre>% s is a serialport object length = 5; resp = read(s,length,"string") resp = "start"</pre>

For more information, see `write` or `read`.

Send a Command

This example shows how to write a terminated SCPI command using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object s.Terminator = "CR/LF" channel = 1; level = 3.44; fprintf(s,"TRIGGER%d:LEVEL2 %1.2f",[channel,level])</pre>	<pre>% s is a serialport object configureTerminator(s,"CR/LF") channel = 1; level = 3.44; write(s,"TRIGGER%d:LEVEL2 %1.2f",[channel,level]) writeline(s,cmd) writeline automatically appends the write terminator.</pre>

For more information, see `configureTerminator` or `writeline`.

Read a Terminated String

This example shows how to perform a terminated string read using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object fprintf(s, "MEASUREMENT:IMMED:TYPE PK2PK") a = fscanf(s, "%e", 6) a = 2.0200</pre> <p>For the format specifier "%e", fscanf returns the terminator and the user must remove it from the string.</p>	<pre>% s is a serialport object writeline(s, "MEASUREMENT:IMMED:TYPE PK2PK") a = readline(s) a = "2.0200" sscanf(a, "%e") a = 2.0200</pre>
<pre>% s is a serial object fprintf(s, "*IDN?") a = fgetl(s) a = 'TEKTRONIX,TDS 210,0,CF:91.1CT FV: v1.16 TDS2</pre> <p>fgetl reads until the specified terminator is reached and then discards the terminator.</p>	<pre>% s is a serialport object writeline(s, "*IDN?") a = readline(s) a = "TEKTRONIX,TDS 210,0,CF:91.1CT FV: v1.16 TDS2</pre> <p>readline reads until the specified terminator is reached and then discards the terminator. There is no option to include the terminator.</p>

Functionality	Use This Instead
<pre>% s is a serial object fprintf(s, "*IDN?") a = fgets(s) a = 'TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04 ' fgets reads until the specified terminator is reached and then returns the terminator.</pre>	

For more information, see `readline`.

Flush Data from Memory

This example shows how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object flushinput(s)</pre>	<pre>% s is a serialport object flush(s, "input")</pre>
<pre>% s is a serial object flushoutput(s)</pre>	<pre>% s is a serialport object flush(s, "output")</pre>
<pre>% s is a serial object flushinput(s) flushoutput(s)</pre>	<pre>% s is a serialport object flush(s)</pre>

For more information, see `flush`.

Set Terminator

This example shows how to set the terminator using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object s.Terminator = "CR/LF";</pre>	<pre>% s is a serialport object configureTerminator(s, "CR/LF")</pre>

Functionality	Use This Instead
<pre>% s is a serial object s.Terminator = {"CR/LF" [10]};</pre>	<pre>% s is a serialport object configureTerminator(s,"CR/LF",10)</pre>

For more information, see `configureTerminator`.

Set Up a Callback Function

This example uses a loopback device to show how to set up a callback function using the recommended functionality.

Functionality	Use This Instead
<pre>s = serial("COM5","BaudRate",115200) s.BytesAvailableFcnCount = 5 s.BytesAvailableFcnMode = "byte" s.BytesAvailableFcn = @instrcallback fopen(s) function instrcallback(src,evt) data = fread(src,src.BytesAvailable) disp(evt) disp(evt.Data) end data = 1 2 3 4 5 Type: 'BytesAvailable' Data: [1x1 struct] AbsTime: [2019 5 2 16 35 9.6710]</pre>	<pre>s = serialport("COM5",115200) configureCallback(s,"byte",5,@instrcallback); function instrcallback(src,evt) data = read(src,src.NumBytesAvailable,"uint8") disp(evt) end data = 1 2 3 4 5 DataAvailableInfo with properties: BytesAvailableFcnCount: 5 AbsTime: 02-May-2019 15:54:09</pre>

For more information, see `configureCallback`.

Read Serial Pin Status

This example shows how to read serial pin status using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object s.PinStatus ans = struct with fields: CarrierDetect: 'on' ClearToSend: 'on' DataSetReady: 'on' RingIndicator: 'on'</pre>	<pre>% s is a serialport object status = getpinstatus(s) status = struct with fields: ClearToSend: 1 DataSetReady: 1 CarrierDetect: 1 RingIndicator: 1</pre>

For more information, see `getpinstatus`.

Set Serial DTR and RTS Pin States

This example shows how to set serial DTR and RTS pin states using the recommended functionality.

Functionality	Use This Instead
<pre>% s is a serial object s.DataTerminalReady = "on";</pre>	<pre>% s is a serialport object setDTR(s,true)</pre>
<pre>% s is a serial object s.RequestToSend = "off";</pre>	<pre>% s is a serialport object setRTS(s,false)</pre>

For more information, see `setDTR` or `setRTS`.

See Also

`serialport` | `serialportlist`

Read Streaming Data from Arduino Using Serial Port Communication

This example shows how to enable callbacks to read streaming ASCII terminated data from an Arduino® Due using the `serialport` interface.

Load Program on the Arduino

Plug in an Arduino Due to your computer.

Load the following program on the Arduino Due using the Arduino IDE. This program writes out continuous points of a sine wave, followed by the "Carriage Return" and "Linefeed" terminators.

```
/*
  SineWavePoints

  Write sine wave points to the serial port, followed by the Carriage Return and LineFeed
  */

int i = 0;

// The setup routine runs once when you press reset:
void setup() {
  // Initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// The loop routine runs over and over again forever:
void loop() {
  // Write the sinewave points, followed by the terminator "Carriage Return" and "LineFeed"
  Serial.print(sin(i*50.0/360.0));
  Serial.write(13);
  Serial.write(10);
  i += 1;
}
```

Establish a Connection to the Arduino

Create a `serialport` instance to connect to your Arduino Due.

Find the serial port that the Arduino is connected to. You can identify the port from the Arduino IDE.

```
serialportlist("available")'  
ans = 3x1 string array  
    "COM1"  
    "COM3"  
    "COM13"
```

Connect to the Arduino Due by creating a `serialport` object using the port and baud rate specified in the Arduino code.

```
arduinoObj = serialport("COM13",9600)  
  
arduinoObj =  
Serialport with properties  
  
        Port: "COM13"  
        BaudRate: 9600  
        NumBytesAvailable: 0  
        NumBytesWritten: 0
```

Show all properties

Prepare the serialport Object to Start Streaming Data

Configure the `serialport` object by clearing old data and configuring its properties.

Set the `Terminator` property to match the terminator that you specified in the Arduino code.

```
configureTerminator(arduinoObj, "CR/LF");
```

Flush the `serialport` object to remove any old data.

```
flush(arduinoObj);
```

Prepare the `UserData` property to store the Arduino data. The `Data` field of the struct saves the sine wave value and the `Count` field saves the x-axis value of the sine wave.

```
arduinoObj.UserData = struct("Data",[], "Count",1)
```

```
arduinoObj =  
Serialport with properties
```

```

        Port: "COM13"
        BaudRate: 9600
    NumBytesAvailable: 10626
    NumBytesWritten: 0

```

Show all properties

Create a callback function `readSineWaveData` that reads the first 1000 ASCII terminated sine wave data points and plots the result.

```

function readSineWaveData(src, ~)

% Read the ASCII data from the serialport object.
data = readline(src);

% Convert the string data to numeric type and save it in the UserData
% property of the serialport object.
src.UserData.Data(end+1) = str2double(data);

% Update the Count value of the serialport object.
src.UserData.Count = src.UserData.Count + 1;

% If 1001 data points have been collected from the Arduino, switch off the
% callbacks and plot the data.
if src.UserData.Count > 1001
    configureCallback(src, "off");
    plot(src.UserData.Data(2:end));
end
end

```

Set the `BytesAvailableFcnMode` property to "terminator" and the `BytesAvailableFcn` property to `@readSineWaveData`. The callback function `readSineWaveData` is triggered when a new sine wave data (with the terminator) is available to be read from the Arduino.

```
configureCallback(arduinoObj, "terminator", @readSineWaveData);
```

The callback function opens the MATLAB figure window with a plot of the first 1000 sine wave data points.

Capabilities and Supported Interfaces and Platforms

In this section...
“What Is the MATLAB Serial Port Interface?” on page 23-50
“Supported Serial Port Interface Standards” on page 23-51
“Supported Platforms” on page 23-51
“Using the Examples with Your Device” on page 23-51
“Find Available Serial Ports Using the serialist Function” on page 23-51

What Is the MATLAB Serial Port Interface?

The MATLAB serial port interface provides direct access to peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port. This interface is established through a serial port object. The serial port object supports functions and properties that allow you to

- Configure serial port communications
- Use serial port control pins
- Write and read data
- Use events and callbacks
- Record information to disk

Instrument Control Toolbox™ software provides additional serial port functionality. In addition to command-line access, this toolbox has a graphical tool called the Test & Measurement Tool, which allows you to communicate with, configure, and transfer data with your serial device without writing code. The Test & Measurement Tool generates MATLAB code for your serial device that you can later reuse to communicate with your device or to develop UI-based applications. The toolbox includes additional serial I/O utility functions that facilitate object creation and configuration, instrument communication, and so on. With the toolbox you can communicate with GPIB- or VISA-compatible instruments.

If you want to communicate with PC-compatible data acquisition hardware such as multifunction I/O boards, you need Data Acquisition Toolbox™ software.

For more information about these products, visit the MathWorks website at <https://www.mathworks.com/products>.

Supported Serial Port Interface Standards

Over the years, several serial port interface standards have been developed. These standards include RS-232, RS-422, and RS-485 - all of which are supported by the MATLAB serial port object. Of these, the most widely used interface standard for connecting computers to peripheral devices is RS-232.

This guide assumes you are using the RS-232 standard, discussed in “Overview of the Serial Port” on page 23-53. Refer to your computer and device documentation to see which interface standard you can use.

Supported Platforms

The MATLAB serial port interface is supported on:

- Linux 64-bit
- macOS 64-bit
- Microsoft Windows 64-bit

Using the Examples with Your Device

Many of the examples in this section reflect specific peripheral devices connected to a serial port — in particular a Tektronix TDS 210 two-channel oscilloscope connected to the COM1 port, on a Windows platform. Therefore, many of the string commands are specific to this instrument and platform.

If you are using a different platform, or your peripheral device is connected to a different serial port, or if it accepts different commands, modify the examples accordingly.

Find Available Serial Ports Using the serialist Function

The `serialist` function returns a list of all serial ports on a system. The list includes virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. This provides a list of the serial ports that you have access to on your computer and could use for serial port communication. For example:

```
serialist
```

```
ans =
```

1×2 string array

"COM1" "COM3"

Overview of the Serial Port

In this section...

“Introduction” on page 23-53

“What Is Serial Communication?” on page 23-53

“The Serial Port Interface Standard” on page 23-53

“Connecting Two Devices with a Serial Cable” on page 23-54

“Serial Port Signals and Pin Assignments” on page 23-55

“Serial Data Format” on page 23-59

“Finding Serial Port Information for Your Platform” on page 23-62

“Using Virtual USB Serial Ports” on page 23-65

“Selected Bibliography” on page 23-65

Introduction

For many serial port applications, you can communicate with your device without detailed knowledge of how the serial port works. If your application is straightforward, or if you are already familiar with the previously mentioned topics, you might want to begin with “The Serial Port Session” on page 23-34 to see how to use your serial port device with MATLAB software.

What Is Serial Communication?

Serial communication is the most common low-level protocol for communicating between two or more devices. Normally, one device is a computer, while the other device can be a modem, a printer, another computer, or a scientific instrument such as an oscilloscope or a function generator.

As the name suggests, the serial port sends and receives bytes of information in a serial fashion — one bit at a time. These bytes are transmitted using either a binary (numerical) format or a text format.

The Serial Port Interface Standard

The serial port interface for connecting two devices is specified by the TIA/EIA-232C standard published by the Telecommunications Industry Association.

The original serial port interface standard was given by RS-232, which stands for Recommended Standard number 232. The term RS-232 is still in popular use, and is used in this guide when referring to a serial communication port that follows the TIA/EIA-232 standard. RS-232 defines these serial port characteristics:

- The maximum bit transfer rate and cable length
- The names, electrical characteristics, and functions of signals
- The mechanical connections and pin assignments

Primary communication is accomplished using three pins: the Transmit Data pin, the Receive Data pin, and the Ground pin. Other pins are available for data flow control, but are not required.

Other standards such as RS-485 define additional functionality such as higher bit transfer rates, longer cable lengths, and connections to as many as 256 devices.

Connecting Two Devices with a Serial Cable

The RS-232 standard defines the two devices connected with a serial cable as the Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE). This terminology reflects the RS-232 origin as a standard for communication between a computer terminal and a modem.

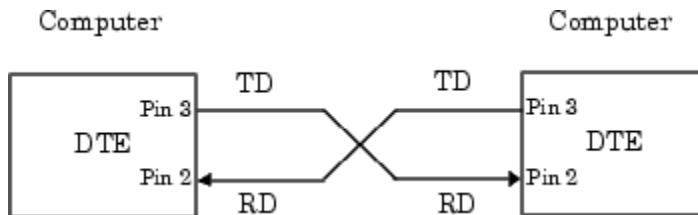
Throughout this guide, your computer is considered a DTE, while peripheral devices such as modems and printers are considered DCEs. Many scientific instruments function as DTEs.

Because RS-232 mainly involves connecting a DTE to a DCE, the pin assignments are defined such that straight-through cabling is used, where pin 1 is connected to pin 1, pin 2 is connected to pin 2, and so on. The following diagram shows a DTE to DCE serial connection using the transmit data (TD) pin and the receive data (RD) pin.

For more information about serial port pins, see “Serial Port Signals and Pin Assignments” on page 23-55.



If you connect two DTEs or two DCEs using a straight serial cable, the TD pins on each device are connected to each other, and the RD pins on each device are connected to each other. Therefore, to connect two like devices, you must use a null modem cable. As shown in the following diagram, null modem cables cross the transmit and receive lines in the cable.

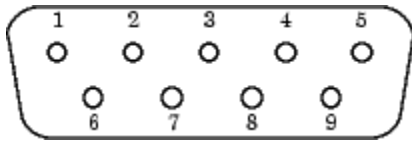


Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/RS-485 adaptor, you can use the MATLAB serial port object with these devices.

Serial Port Signals and Pin Assignments

Serial ports consist of two signal types: data signals and control signals. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most Windows and UNIX platforms use a 9-pin connection. In fact, only three pins are required for serial port communications: one for receiving data, one for transmitting data, and one for the signal ground.

The following diagram shows the pin assignment scheme for a 9-pin male connector on a DTE.



The pins and signals associated with the 9-pin connector are described in the following table. Refer to the RS-232 standard for a description of the signals and pin assignments used for a 25-pin connector.

Serial Port Pin and Signal Assignments

Pin	Label	Signal Name	Signal Type
1	CD	Carrier Detect	Control
2	RD	Received Data	Data
3	TD	Transmitted Data	Data
4	DTR	Data Terminal Ready	Control
5	GND	Signal Ground	Ground
6	DSR	Data Set Ready	Control
7	RTS	Request to Send	Control
8	CTS	Clear to Send	Control
9	RI	Ring Indicator	Control

The term data set is synonymous with modem or device, while the term data terminal is synonymous with computer.

Note The serial port pin and signal assignments are with respect to the DTE. For example, data is transmitted from the TD pin of the DTE to the RD pin of the DCE.

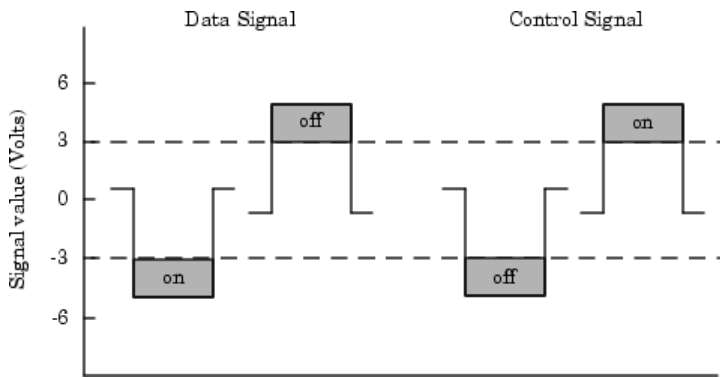
Signal States

Signals can be in either an active state or an inactive state. An active state corresponds to the binary value 1, while an inactive state corresponds to the binary value 0. An active signal state is often described as logic 1, on, true, or a mark. An inactive signal state is often described as logic 0, off, false, or a space.

For data signals, the **on** state occurs when the received signal voltage is more negative than -3 volts, while the **off** state occurs for voltages more positive than 3 volts. For control signals, the **on** state occurs when the received signal voltage is more positive than 3 volts, while the **off** state occurs for voltages more negative than -3 volts. The voltage between -3 volts and +3 volts is considered a transition region, and the signal state is undefined.

To bring the signal to the **on** state, the controlling device unasserts (or lowers) the value for data pins and asserts (or raises) the value for control pins. Conversely, to bring the signal to the **off** state, the controlling device asserts the value for data pins and unasserts the value for control pins.

The following diagram shows the **on** and **off** states for a data signal and for a control signal.



The Data Pins

Most serial port devices support full-duplex communication meaning that they can send and receive data at the same time. Therefore, separate pins are used for transmitting and receiving data. For these devices, the TD, RD, and GND pins are used. However, some types of serial port devices support only one-way or half-duplex communications. For these devices, only the TD and GND pins are used. This guide assumes that a full-duplex serial port is connected to your device.

The TD pin carries data transmitted by a DTE to a DCE. The RD pin carries data that is received by a DTE from a DCE.

The Control Pins

The control pins of a 9-pin serial port are used to determine the presence of connected devices and control the flow of data. The control pins include

- “The RTS and CTS Pins” on page 23-58
- “The DTR and DSR Pins” on page 23-58
- “The CD and RI Pins” on page 23-59

The RTS and CTS Pins

The RTS and CTS pins are used to signal whether the devices are ready to send or receive data. This type of data flow control—called hardware handshaking—is used to prevent data loss during transmission. When enabled for both the DTE and DCE, hardware handshaking using RTS and CTS follows these steps:

- 1 The DTE asserts the RTS pin to instruct the DCE that it is ready to receive data.
- 2 The DCE asserts the CTS pin indicating that it is clear to send data over the TD pin. If data can no longer be sent, the CTS pin is unasserted.
- 3 The data is transmitted to the DTE over the TD pin. If data can no longer be accepted, the RTS pin is unasserted by the DTE and the data transmission is stopped.

To enable hardware handshaking in MATLAB software, see “Controlling the Flow of Data: Handshaking” on page 23-102.

The DTR and DSR Pins

Many devices use the DSR and DTR pins to signal if they are connected and powered. Signaling the presence of connected devices using DTR and DSR follows these steps:

- 1 The DTE asserts the DTR pin to request that the DCE connect to the communication line.
- 2 The DCE asserts the DSR pin to indicate it is connected.
- 3 DCE unasserts the DSR pin when it is disconnected from the communication line.

DTR/DSR lines have multiple purposes depending on the application. For example these lines can be used for handshaking, flow control, or even providing power to the unit. However RTS/CTS lines are mostly used for data flow control between the host and device. Refer to your device documentation to determine its specific pin behavior.

The CD and RI Pins

The CD and RI pins are typically used to indicate the presence of certain signals during modem-modem connections.

A modem uses a CD pin to signal that it has made a connection with another modem, or has detected a carrier tone. CD is asserted when the DCE is receiving a signal of a suitable frequency. CD is unasserted if the DCE is not receiving a suitable signal.

The RI pin is used to indicate the presence of an audible ringing signal. RI is asserted when the DCE is receiving a ringing signal. RI is unasserted when the DCE is not receiving a ringing signal (e.g., it is between rings).

Serial Data Format

The serial data format includes one start bit, between five and eight data bits, and one stop bit. A parity bit and an additional stop bit might be included in the format as well. The following diagram illustrates the serial data format.



The following notation expresses the format for serial port data:

number of data bits - parity type - number of stop bits

For example, 8-N-1 is interpreted as eight data bits, no parity bit, and one stop bit, while 7-E-2 is interpreted as seven data bits, even parity, and two stop bits.

The data bits are often referred to as a character because these bits usually represent an ASCII character. The remaining bits are called framing bits because they frame the data bits.

Bytes Versus Values

A byte is the collection of bits that comprise the serial data format. At first, this term might seem inaccurate because a byte is 8 bits and the serial data format can range between 7 bits and 12 bits. However, when serial data is stored on your computer, the framing bits are stripped away, and only the data bits are retained. Moreover, eight data

bits are always used regardless of the number of data bits specified for transmission, with the unused bits assigned a value of 0.

When reading or writing data, you might need to specify a value, which can consist of one or more bytes. For example, if you read one value from a device using the `int32` format, that value consists of four bytes. For more information about reading and writing values, see “Write and Read Data” on page 23-73.

Synchronous and Asynchronous Communication

The RS-232 standard supports two types of communication protocols: synchronous and asynchronous.

Using the synchronous protocol, all transmitted bits are synchronized to a common clock signal. The two devices initially synchronize themselves to each other, and continually send characters to stay synchronized. Even when actual data is not really being sent, a constant flow of bits allows each device to know where the other is at any given time. That is, each bit that is sent is either actual data or an idle character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

Using the asynchronous protocol, each device uses its own internal clock, resulting in bytes that are transferred at arbitrary times. So, instead of using time as a way to synchronize the bits, the data format is used.

In particular, the data transmission is synchronized using the start bit of the word, while one or more stop bits indicate the end of the word. The requirement to send these additional bits causes asynchronous communications to be slightly slower than synchronous. However, it has the advantage that the processor does not have to deal with the additional idle characters. Most serial ports operate asynchronously.

Note When used in this guide, the terms synchronous and asynchronous refer to whether read or write operations block access to the MATLAB command line. For more information, see “Controlling Access to the MATLAB Command Line” on page 23-74.

How Are the Bits Transmitted?

By definition, serial data is transmitted one bit at a time. The order in which the bits are transmitted is:

- 1 The start bit is transmitted with a value of 0.
- 2 The data bits are transmitted. The first data bit corresponds to the least significant bit (LSB), while the last data bit corresponds to the most significant bit (MSB).
- 3 The parity bit (if defined) is transmitted.
- 4 One or two stop bits are transmitted, each with a value of 1.

The baud rate is the number of bits transferred per second. The transferred bits include the start bit, the data bits, the parity bit (if defined), and the stop bits.

Start and Stop Bits

As described in “Synchronous and Asynchronous Communication” on page 23-60, most serial ports operate asynchronously. This means that the transmitted byte must be identified by start and stop bits. The start bit indicates when the data byte is about to begin; the stop bit(s) indicate(s) when the data byte has been transferred. The process of identifying bytes with the serial data format follows these steps:

- 1 When a serial port pin is idle (not transmitting data), it is in an **on** state.
- 2 When data is about to be transmitted, the serial port pin switches to an **off** state due to the start bit.
- 3 The serial port pin switches back to an **on** state due to the stop bit(s). This indicates the end of the byte.

Data Bits

The data bits transferred through a serial port might represent device commands, sensor readings, error messages, and so on. The data can be transferred as either binary data or ASCII data.

Most serial ports use between five and eight data bits. Binary data is typically transmitted as eight bits. Text-based data is transmitted as either seven bits or eight bits. If the data is based on the ASCII character set, a minimum of seven bits is required because there are 2^7 or 128 distinct characters. If an eighth bit is used, it must have a value of 0. If the data is based on the extended ASCII character set, eight bits must be used because there are 2^8 or 256 distinct characters.

The Parity Bit

The parity bit provides simple error (parity) checking for the transmitted data. The following table shows the types of parity checking.

Parity Types

Parity Type	Description
Even	The data bits plus the parity bit result in an even number of 1s.
Mark	The parity bit is always 1.
Odd	The data bits plus the parity bit result in an odd number of 1s.
Space	The parity bit is always 0.

Mark and space parity checking are seldom used because they offer minimal error detection. You might choose to not use parity checking at all.

The parity checking process follows these steps:

- 1 The transmitting device sets the parity bit to 0 or to 1, depending on the data bit values and the type of parity-checking selected.
- 2 The receiving device checks if the parity bit is consistent with the transmitted data. If it is, the data bits are accepted. If it is not, an error is returned.

Note Parity checking can detect only 1-bit errors. Multiple-bit errors can appear as valid data.

For example, suppose the data bits 01110001 are transmitted to your computer. If even parity is selected, the parity bit is set to 0 by the transmitting device to produce an even number of 1s. If odd parity is selected, the parity bit is set to 1 by the transmitting device to produce an odd number of 1s.

Finding Serial Port Information for Your Platform

This section describes the ways to find serial port information for Windows and UNIX platforms.

Note Your operating system provides default values for all serial port settings. However, these settings are overridden by your MATLAB code, and will have no effect on your serial port application.

Use the seriallist Function

The `seriallist` function returns a list of all serial ports on a system. The list includes virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. This provides a list of the serial ports that you have access to on your computer and could use for serial port communication. For example:

```
seriallist
```

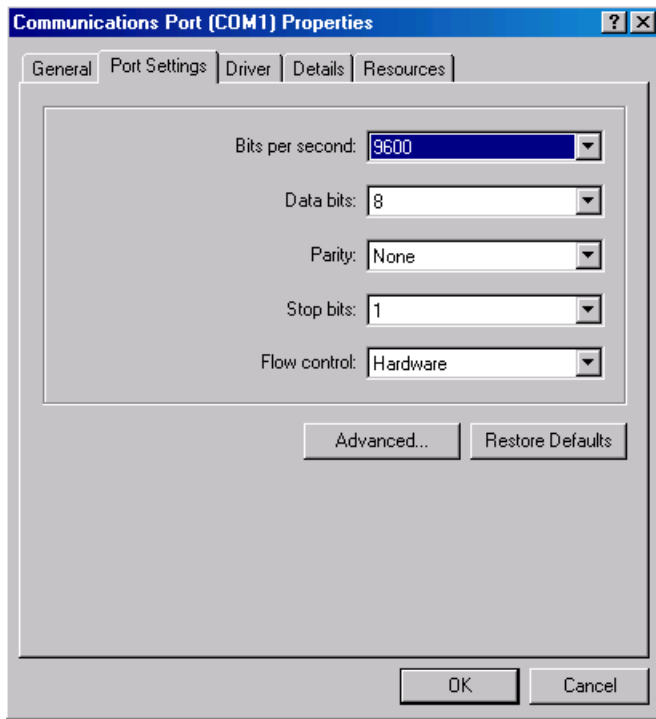
```
ans =
```

```
    1x2 string array
```

```
    "COM1"    "COM3"
```

Microsoft Windows Platform

- 1 Open the Windows Control Panel.
- 2 Click **Device Manager**.
- 3 In the **Device Manager** dialog box, expand the Ports node.
- 4 Double-click the Communications Port (COM1) node.
- 5 Select the **Port Settings** tab.



UNIX Platform

To find serial port information for UNIX platforms, you need to know the serial port names. These names might vary between different operating systems.

On a Linux platform, serial port devices are typically named `ttyS0`, `ttyS1`, etc. Use the `setserial` command to display or configure serial port information. For example, to display which ports are available:

```
setserial -bg /dev/ttyS*
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

To display detailed information about `ttyS0`:

```
setserial -ag /dev/ttyS0
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
      Baud_base: 115200, close_delay: 50, divisor: 0
```

```
closing_wait: 3000, closing_wait2: infinte
Flags: spd_normal skip_test session_lockout
```

Note If the `setserial -ag` command does not work, make sure you have read and write permission for the port.

For all supported UNIX platforms, use the `stty` command to display or configure serial port information. For example, to display serial port properties for `ttyS0`, enter:

```
stty -a < /dev/ttyS0
```

To configure the baud rate to 4800 bits per second, enter:

```
stty speed 4800 < /dev/ttyS0 > /dev/ttyS0
```

Using Virtual USB Serial Ports

If you have devices that present themselves as serial ports on your operating system, you can use them as virtual USB serial ports in MATLAB. An example of such devices would be a USB Serial Dongle. For Bluetooth devices, use the Bluetooth support in the Instrument Control Toolbox. See “Bluetooth Interface Overview” (Instrument Control Toolbox) for more information.

MATLAB can communicate with these devices as long as the serial drivers provided by the device vendor are able to emulate the native hardware. Certain software, like HyperTerminal, does not require the device driver to fully implement and support the native hardware.

Selected Bibliography

- [1] TIA/EIA-232-F, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*.
- [2] Jan Axelson, *Serial Port Complete*, Lakeview Research, Madison, WI, 1998.
- [3] *Instrument Communication Handbook*, IOTech, Inc., Cleveland, OH, 1991.
- [4] *TDS 200-Series Two Channel Digital Oscilloscope Programmer Manual*, Tektronix, Inc., Wilsonville, OR.

[5] *Courier High Speed Modems User's Manual*, U.S. Robotics, Inc., Skokie, IL, 1994.

Create a Serial Port Object

In this section...

- “Overview of a Serial Port Object” on page 23-67
- “Configuring Properties During Object Creation” on page 23-69
- “The Serial Port Object Display” on page 23-69
- “Creating an Array of Serial Port Objects” on page 23-70

Overview of a Serial Port Object

The `serial` function requires the name of the serial port connected to your device as an input argument. Additionally, you can configure property values during object creation. To create a serial port object associated with the serial port enter:

```
s = serial('port');
```

This creates a serial port object associated with the serial port specified by `'port'`. If `'port'` does not exist, or if it is in use, you will not be able to connect the serial port object to the device. `'port'` object name will depend upon the platform that the serial port is on.

The `seriallist` function returns a list of all serial ports on a system. The list includes virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. This provides a list of the serial ports that you have access to on your computer and could use for serial port communication. For example:

```
seriallist
ans =
    1x2 string array
    "COM1"    "COM3"
```

This table shows an example of serial constructors on different platforms:

Platform	Serial Constructor
Linux 64-bit	<code>serial('/dev/ttyS0');</code>

Platform	Serial Constructor
macOS 64-bit	<code>serial('/dev/tty.KeySerial1');</code>
Microsoft Windows 64-bit	<code>serial('com1');</code>

The serial port object `s` now exists in the MATLAB workspace. To display the class of `s`, use the `whos` command.

```
whos s
  Name      Size      Bytes  Class
  s         1x1         512   serial object
```

Grand total is 11 elements using 512 bytes

Note The first time you try to access a serial port in MATLAB using the `s = serial('port')` call, make sure that the port is free and is not already open in any other application. If the port is open in another application, MATLAB cannot access it. Once you have accessed in MATLAB, you can open the same port in other applications and MATLAB will continue to use it along with any other application that has it open as well.

Once the serial port object is created, the following properties are automatically assigned values. These general-purpose properties provide descriptive information about the serial port object based on the object type and the serial port.

Descriptive General Purpose Properties

Property Name	Description
Name	Specify a descriptive name for the serial port object
"port"	Indicate the platform-specific serial port name
Type	Indicate the object type

Display the values of these properties for `s` with the `get` function. On a Windows platform, it will look like this:

```
get(s,{'Name','Port','Type'})
ans =
    'Serial-COM1'    'COM1'    'serial'
```

Configuring Properties During Object Creation

You can configure serial port properties during object creation. `serial` accepts property names and property values in the same format as the `set` function. For example, you can specify property name/property value pairs.

```
s = serial('port', 'BaudRate', 4800, 'Parity', 'even');
```

If you specify an invalid property name, the object is not created. However, if you specify an invalid value for some properties (for example, `BaudRate` is set to 50), the object might be created but you are not informed of the invalid value until you connect the object to the device with the `fopen` function.

The Serial Port Object Display

The serial port object provides you with a convenient display that summarizes important configuration and state information. You can invoke the display summary these three ways:

- Type the serial port object variable name at the command line.
- Exclude the semicolon when creating a serial port object.
- Exclude the semicolon when configuring properties using the dot notation.

To display summary information using the Workspace browser, right-click an instrument object and select **Display Summary** from the context menu.

The display summary for the serial port object `s` on a Windows platform is:

```
Serial Port Object : Serial-COM1
```

Communication Settings

```
Port:          COM1
BaudRate:     9600
Terminator:   'LF'
```

Communication State

```
Status:       closed
RecordStatus: off
```

Read/Write State

```
TransferStatus: idle
BytesAvailable: 0
```

```
ValuesReceived:    0
ValuesSent:        0
```

Creating an Array of Serial Port Objects

In MATLAB, you create an array from existing variables by concatenating those variables together. The same is true for serial port objects. For example, suppose you create the serial port objects `s1` and `s2` on a Windows platform.

```
s1 = serial('COM1');
s2 = serial('COM2');
```

Create a serial port object array, consisting of `s1` and `s2`, using the usual MATLAB syntax. To create the row array `x`, enter:

```
x = [s1 s2]
```

Instrument Object Array

Index:	Type:	Status:	Name:
1	serial	closed	Serial-COM1
2	serial	closed	Serial-COM2

To create the column array `y`, enter:

```
y = [s1;s2];
```

Note that you cannot create a matrix of serial port objects. For example, you cannot create the matrix:

```
z = [s1 s2;s1 s2];
??? Error using ==> serial/vertcat
Only a row or column vector of instrument objects can be created.
```

Depending on your application, you might want to pass an array of serial port objects to a function. For example, to configure the baud rate and parity for `s1` and `s2` using one call to `set`:

```
set(x, 'BaudRate', 19200, 'Parity', 'even')
```

Connect to the Device

Before using the serial port object to write or read data, you must connect it to your device via the serial port specified in the `serial` function. You connect a serial port object to the device with the `fopen` function.

```
fopen(s)
```

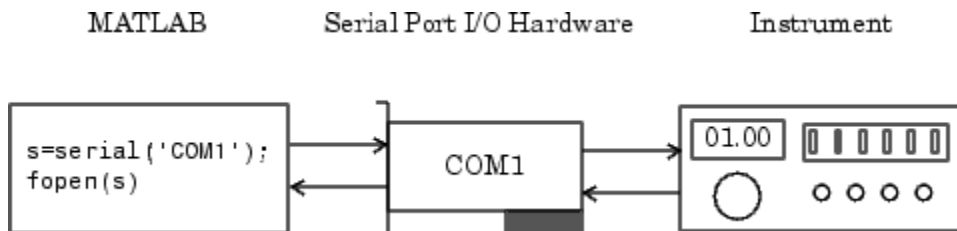
Some properties are read only while the serial port object is connected and must be configured before using `fopen`. Examples include the “InputBufferSize” and the “OutputBufferSize” properties. To determine when you can configure a property, see “Property Reference” on page 23-115.

Note You can create any number of serial port objects, but you can connect only one serial port object per MATLAB session to a given serial port at a time. However, the serial port is not locked by the session, so other applications or other instances of MATLAB software can access the same serial port, which could result in a conflict, with unpredictable results.

To verify that the serial port object is connected to the device, examine the `Status` property.

```
s.Status
ans =
open
```

As shown in the following illustration, the connection between the serial port object and the device is complete; data is readable and writable.



Configure Communication Settings

Before you can write or read data, both the serial port object and the device must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the serial data format on page 23-59. The following table describes these properties.

Communication Properties

Property Name	Description
"BaudRate"	Specify the rate at which bits are transmitted
"DataBits"	Specify the number of data bits to transmit
"Parity"	Specify the type of parity checking
"StopBits"	Specify the number of bits used to indicate the end of a byte
"Terminator"	Specify the terminator character

Note If the serial port object and the device communication settings are not identical, data is not readable or writable.

Refer to the device documentation for an explanation of its supported communication settings.

Write and Read Data

In this section...

“Before You Begin” on page 23-73

“Example — Introduction to Writing and Reading Data” on page 23-73

“Controlling Access to the MATLAB Command Line” on page 23-74

“Writing Data” on page 23-75

“Reading Data” on page 23-80

“Example — Writing and Reading Text Data” on page 23-86

“Example — Parsing Input Data Using textscan” on page 23-87

“Example — Reading Binary Data” on page 23-88

Before You Begin

For many serial port applications, there are three important questions that you should consider when writing or reading data:

- Will the read or write function block access to the MATLAB command line?
- Is the data to be transferred binary (numerical) or text?
- Under what conditions will the read or write operation complete?

For write operations, these questions are answered in “Writing Data” on page 23-75. For read operations, these questions are answered in “Reading Data” on page 23-80.

Example — Introduction to Writing and Reading Data

Suppose you want to return identification information for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1 on a Windows platform. This requires writing the `*IDN?` command to the instrument using the `fprintf` function, and reading back the result of that command using the `fscanf` function.

```
s = serial('COM1');  
fopen(s)  
fprintf(s, '*IDN?')  
out = fscanf(s)
```

The resulting identification information is:

```
out =  
TEKTRONIX, TDS 210, 0, CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

End the serial port session.

```
fclose(s)  
delete(s)  
clear s
```

Controlling Access to the MATLAB Command Line

You control access to the MATLAB command line by specifying whether a read or write operation is *synchronous* or *asynchronous*.

A synchronous operation blocks access to the command line until the read or write function completes execution. An asynchronous operation does not block access to the command line, and you can issue additional commands while the read or write function executes in the background.

The terms *synchronous* and *asynchronous* are often used to describe how the serial port operates at the hardware level. The RS-232 standard supports an asynchronous communication protocol. Using this protocol, each device uses its own internal clock. The data transmission is synchronized using the start bit of the bytes, while one or more stop bits indicate the end of the byte. For more information on start bits and stop bits, see “Serial Data Format” on page 23-59. The RS-232 standard also supports a synchronous mode where all transmitted bits are synchronized to a common clock signal.

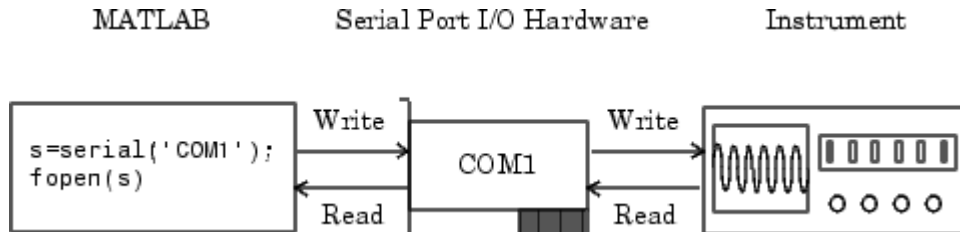
At the hardware level, most serial ports operate asynchronously. However, using the default behavior for many of the read and write functions, you can mimic the operation of a synchronous serial port.

Note When used in this guide, the terms *synchronous* and *asynchronous* refer to whether read or write operations block access to the MATLAB command-line. In other words, these terms describe how the software behaves, and not how the hardware behaves.

The two main advantages of writing or reading data asynchronously are:

- You can issue another command while the write or read function is executing.
- You can use all supported callback properties (see “Events and Callbacks” on page 23-91).

For example, because serial ports have separate read and write pins, you can simultaneously read and write data. This is illustrated in the following diagram.



Writing Data

This section describes writing data to your serial port device in three parts:

- “The Output Buffer and Data Flow” on page 23-76 describes the flow of data from MATLAB software to the device.
- “Writing Text Data” on page 23-77 describes how to write text data (string commands) to the device.
- “Writing Binary Data” on page 23-79 describes how to write binary (numerical) data to the device.

The following table shows the functions associated with writing data.

Functions Associated with Writing Data

Function Name	Description
<code>fprintf</code>	Write text to the device
<code>fwrite</code>	Write binary data to the device
<code>stopasync</code>	Stop asynchronous read and write operations

The following table shows the properties associated with writing data.

Properties Associated with Writing Data

Property Name	Description
"BytesToOutput"	Number of bytes currently in the output buffer
"OutputBufferSize"	Size of the output buffer in bytes
"Timeout"	Waiting time to complete a read or write operation
"TransferStatus"	Indicate if an asynchronous read or write operation is in progress
"ValuesSent"	Total number of values written to the device

The Output Buffer and Data Flow

The output buffer is computer memory allocated by the serial port object to store data that is to be written to the device. When writing data to your device, the data flow follows these two steps:

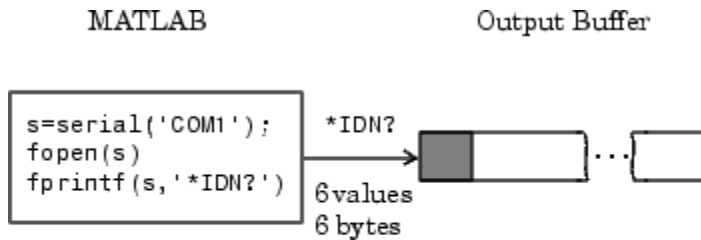
- 1 The data specified by the write function is sent to the output buffer.
- 2 The data in the output buffer is sent to the device.

The `OutputBufferSize` property specifies the maximum number of bytes that you can store in the output buffer. The `BytesToOutput` property indicates the number of bytes currently in the output buffer. The default values for these properties are:

```
s = serial('COM1');
get(s,{'OutputBufferSize','BytesToOutput'})
ans =
    [512]    [0]
```

If you attempt to write more data than can fit in the output buffer, an error is returned and no data is written.

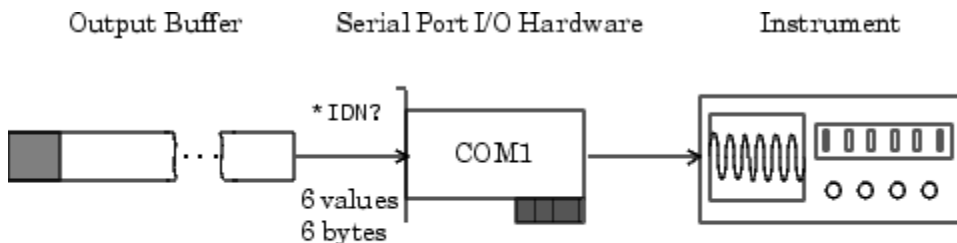
For example, suppose you write the string command `*IDN?` to the TDS 210 oscilloscope using the `fprintf` function. As shown in the following diagram, the string is first written to the output buffer as six values.



- Bytes used during write
- Bytes unused during write

The `*IDN?` command consists of six values because the terminator is automatically written. Moreover, the default data format for the `fprintf` function specifies that one value corresponds to one byte. For more information about bytes and values, see “Bytes Versus Values” on page 23-59. `fprintf` and the terminator are discussed in “Writing Text Data” on page 23-77.

As shown in the following diagram, after the string is written to the output buffer, it is then written to the device via the serial port.



- Bytes used during write
- Bytes unused during write

Writing Text Data

You use the `fprintf` function to write text data to the device. For many devices, writing text data means writing string commands that change device settings, prepare the device to return data or status information, and so on.

For example, the `Display:Contrast` command changes the display contrast of the oscilloscope.

```
s = serial('COM1');  
fopen(s)  
fprintf(s, 'Display:Contrast 45')
```

By default, `fprintf` writes data using the `%s\n` format because many serial port devices accept only text-based commands. However, you can specify many other formats, as described in the `fprintf` reference pages.

To verify the number of values sent to the device, use the `ValuesSent` property.

```
s.ValuesSent  
ans =  
    20
```

Note that the `ValuesSent` property value includes the terminator because each occurrence of `\n` in the command sent to the device is replaced with the `Terminator` property value.

```
s.Terminator  
ans =  
LF
```

The default value of `Terminator` is the linefeed character. The terminator required by your device will be described in its documentation.

Synchronous Versus Asynchronous Write Operations

By default, `fprintf` operates synchronously and blocks the MATLAB command line until execution completes. To write text data asynchronously to the device, you must specify `async` as the last input argument to `fprintf`.

```
fprintf(s, 'Display:Contrast 45', 'async')
```

Asynchronous operations do not block access to the MATLAB command line. Additionally, while an asynchronous write operation is in progress, you can:

- Execute an asynchronous read operation because serial ports have separate pins for reading and writing
- Make use of all supported callback properties

To determine which asynchronous operations are in progress, use the `TransferStatus` property. If no asynchronous operations are in progress, `TransferStatus` is `idle`.

```
s.TransferStatus  
ans =  
idle
```

Completing a Write Operation with `fprintf`

A synchronous or asynchronous write operation using `fprintf` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Stop an asynchronous write operation with the `stopasync` function.

Rules for Writing the Terminator

The `Terminator` property value replaces all occurrences of `\n` in `cmd`. Therefore, when you use the default format `%s\n`, all commands written to the device end with this property value. Refer to your device documentation for the terminator required by your device.

Writing Binary Data

You use the `fwrite` function to write binary data to the device. Writing binary data means writing numerical values. A typical application for writing binary data involves writing calibration data to an instrument such as an arbitrary waveform generator.

Note Some serial port devices accept only text-based commands. These commands might use the SCPI language or some other vendor-specific language. Therefore, you might need to use the `fprintf` function for all write operations.

By default, `fwrite` translates values using the `uchar` precision. However, you can specify many other precisions as described in the reference pages for this function.

By default, `fwrite` operates synchronously. To write binary data asynchronously to the device, you must specify `async` as the last input argument to `fwrite`. For more information about synchronous and asynchronous write operations, see “Writing Text Data” on page 23-77. For a description of the rules used by `fwrite` to complete a write operation, refer to its reference pages.

Troubleshooting Common Errors

Use this table to identify common `fprintf` errors.

Error	Occurs when	Troubleshooting
<p>??? Error using ==> serial.fwrite at 199 OBJ must be connected to the hardware with FOPEN.</p>	<p>You perform a write operation and the serial port object is not connected to the device.</p>	<p>Use <code>fopen</code> to establish a connection to the device.</p>
<p>??? Error using ==> serial.fwrite at 199 The number of bytes written must be less than or equal to <code>OutputBufferSize-BytesToOutput</code>.</p>	<p>The output buffer is not able to hold all the data to be written.</p>	<p>Specify the size of the output buffer with the <code>OutputBufferSize</code> property.</p>
<p>??? Error using ==> serial.fwrite at 192 FWRITE cannot be called. The <code>FlowControl</code> property is set to 'hardware' and the Clear To Send (CTS) pin is high. This could indicate that the serial device may not be turned on, may not be connected, or does not use hardware handshaking</p>	<ul style="list-style-type: none"> • You set the <code>FlowControl</code> property on a serial object to hardware. • The device is either not connected or a connected device is not asserting that is ready to receive data. 	<p>Check your remote device status and flow control settings to see if hardware flow control is causing MATLAB errors.</p>

Reading Data

This section describes reading data from your serial port device in three parts:

- “The Input Buffer and Data Flow” on page 23-81 describes the flow of data from the device to MATLAB software.
- “Reading Text Data” on page 23-83 describes how to read from the device, and format the data as text.

- “Reading Binary Data” on page 23-85 describes how to read binary (numerical) data from the device.

The following table shows the functions associated with reading data.

Functions Associated with Reading Data

Function Name	Description
<code>fgetl</code>	Read one line of text from the device and discard the terminator
<code>fgets</code>	Read one line of text from the device and include the terminator
<code>fread</code>	Read binary data from the device
<code>fscanf</code>	Read data from the device and format as text
<code>readasync</code>	Read data asynchronously from the device
<code>stopasync</code>	Stop asynchronous read and write operations

The following table shows the properties associated with reading data.

Properties Associated with Reading Data

Property Name	Description
“BytesAvailable”	Number of bytes available in the input buffer
“InputBufferSize”	Size of the input buffer in bytes
“ReadAsyncMode”	Specify whether an asynchronous read operation is continuous or manual
“Timeout”	Waiting time to complete a read or write operation
“TransferStatus”	Indicate if an asynchronous read or write operation is in progress
“ValuesReceived”	Total number of values read from the device

The Input Buffer and Data Flow

The input buffer is computer memory allocated by the serial port object to store data that is to be read from the device. When reading data from your device, the data flow follows these two steps:

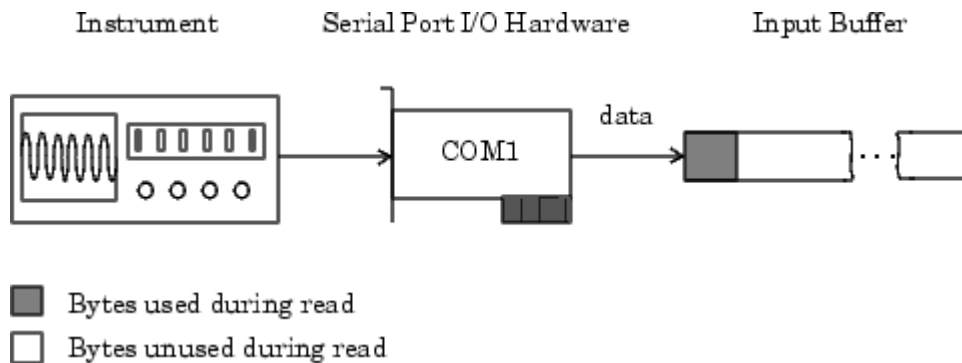
- 1 The data read from the device is stored in the input buffer.
- 2 The data in the input buffer is returned to the MATLAB variable specified by the read function.

The `InputBufferSize` property specifies the maximum number of bytes you can store in the input buffer. The `BytesAvailable` property indicates the number of bytes currently available to be read from the input buffer. The default values for these properties are:

```
s = serial('COM1');
get(s,{'InputBufferSize','BytesAvailable'})
ans =
    [512]    [0]
```

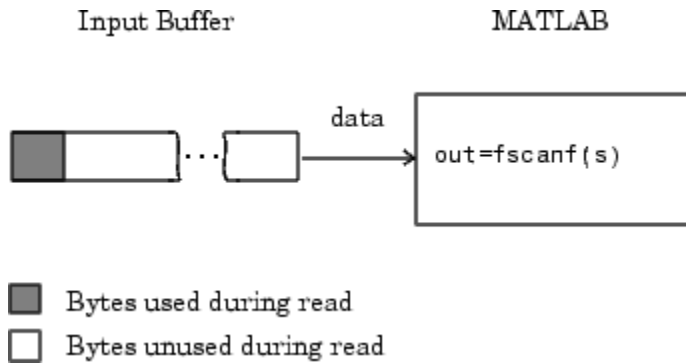
If you attempt to read more data than can fit in the input buffer, an error is returned and no data is read.

For example, suppose you use the `fscanf` function to read the text-based response of the `*IDN?` command previously written to the TDS 210 oscilloscope. As shown in the following diagram, the text data is first read into the input buffer via the serial port.



Note that for a given read operation, you might not know the number of bytes returned by the device. Therefore, you might need to preset the `InputBufferSize` property to a sufficiently large value before connecting the serial port object.

As shown in the following diagram, after the data is stored in the input buffer, it is then transferred to the output variable specified by `fscanf`.



Reading Text Data

You use the `fgetl`, `fgets`, and `fscanf` functions to read data from the device, and format the data as text.

For example, suppose you want to return identification information for the oscilloscope. This requires writing the `*IDN?` command to the instrument, and then reading back the result of that command.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
out = fscanf(s)
out =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

By default, `fscanf` reads data using the `%c` format because the data returned by many serial port devices is text based. However, you can specify many other formats as described in the `fscanf` reference pages.

To verify the number of values read from the device—including the terminator, use the `ValuesReceived` property.

```
s.ValuesReceived
ans =
    56
```

Synchronous Versus Asynchronous Read Operations

You specify whether read operations are synchronous or asynchronous with the `ReadAsyncMode` property. You can configure `ReadAsyncMode` to `continuous` or `manual`.

If `ReadAsyncMode` is `continuous` (the default value), the serial port object continuously queries the device to determine if data is available to be read. If data is available, it is asynchronously stored in the input buffer. To transfer the data from the input buffer to MATLAB, use one of the synchronous (blocking) read functions such as `fgetl` or `fscanf`. If data is available in the input buffer, these functions return quickly.

```
s.ReadAsyncMode = 'continuous';
fprintf(s, '*IDN?')
s.BytesAvailable
ans =
    56
out = fscanf(s);
```

If `ReadAsyncMode` is `manual`, the serial port object does not continuously query the device to determine if data is available to be read. To read data asynchronously, use the `readasync` function. Then use one of the synchronous read functions to transfer data from the input buffer to MATLAB.

```
s.ReadAsyncMode = 'manual';
fprintf(s, '*IDN?')
s.BytesAvailable
ans =
    0
readasync(s)
s.BytesAvailable
ans =
    56
out = fscanf(s);
```

Asynchronous operations do not block access to the MATLAB command line. Additionally, while an asynchronous read operation is in progress, you can:

- Execute an asynchronous write operation because serial ports have separate pins for reading and writing
- Make use of all supported callback properties

To determine which asynchronous operations are in progress, use the `TransferStatus` property. If no asynchronous operations are in progress, then `TransferStatus` is `idle`.

```
s.TransferStatus
ans =
idle
```

Rules for Completing a Read Operation with fscanf

A read operation with `fscanf` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of values specified is read.
- The input buffer is filled.

Reading Binary Data

You use the `fread` function to read binary data from the device. Reading binary data means that you return numerical values to MATLAB.

For example, suppose you want to return the cursor and display settings for the oscilloscope. This requires writing the `CURSOR?` and `DISPLAY?` commands to the instrument, and then reading back the results of those commands.

```
s = serial('COM1');
fopen(s)
fprintf(s,'CURSOR?')
fprintf(s,'DISPLAY?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the device. To verify the number of values read, use the `BytesAvailable` property.

```
s.BytesAvailable
ans =
    69
```

To return the data to MATLAB, use any of the synchronous read functions. However, if you use `fgetl`, `fgets`, or `fscanf`, you must issue the function twice because there are two terminators stored in the input buffer. To return all the data to MATLAB in one function call, use `fread`.

```
out = fread(s,69);
```

By default, `fread` returns numerical values in double precision arrays. However, you can specify many other precisions as described in the `fread` reference pages. To convert the numerical data to text, use the MATLAB `char` function.

```
val = char(out)'  
val =  
HBARS;CH1;SECONDS;-1.0E-3;1.0E-3;VOLTS;-6.56E-1;6.24E-1  
YT;DOTS;0;45
```

For more information about synchronous and asynchronous read operations, see “Reading Text Data” on page 23-83. For a description of the rules used by `fread` to complete a read operation, refer to its reference pages.

Example — Writing and Reading Text Data

This example illustrates how to communicate with a serial port instrument by writing and reading text data.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the COM1 port. Therefore, many of the following commands are specific to this instrument. A sine wave is input into channel 2 of the oscilloscope, and your job is to measure the peak-to-peak voltage of the input signal.

- 1 Create a serial port object — Create the serial port object `s` associated with serial port COM1.

```
s = serial('COM1');
```

- 2 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

- 3 Write and read data — Write the `*IDN?` command to the instrument using `fprintf`, and then read back the result of the command using `fscanf`.

```
fprintf(s,'*IDN?')  
idn = fscanf(s)  
idn =  
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

You need to determine the measurement source. Possible measurement sources include channel 1 and channel 2 of the oscilloscope.

```
fprintf(s, 'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s)
source =
CH1
```

The scope is configured to return a measurement from channel 1. Because the input signal is connected to channel 2, you must configure the instrument to return a measurement from this channel.

```
fprintf(s, 'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(s, 'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s)
source =
CH2
```

Configure the scope to return the peak-to-peak voltage, and then request the value of this measurement.

```
fprintf(s, 'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s, 'MEASUREMENT:MEAS1:VALUE?')
```

Transfer data from the input buffer to MATLAB using `fscanf`.

```
ptop = fscanf(s, '%g')
ptop =
2.0199999809E0
```

- 4 Disconnect and clean up — When you no longer need `s` disconnect it from the instrument and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Example — Parsing Input Data Using `textscan`

This example illustrates how to use the `textscan` function to parse and format data that you read from a device. `textscan` is particularly useful when you want to parse a string into one or more variables, where each variable has its own specified format.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

- 1 Create a serial port object — Create the serial port object `s` associated with serial port COM1.

```
s = serial('COM1');
```

- 2 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

- 3 Write and read data — Write the `RS232?` command to the instrument using `fprintf`, and then read back the result of the command using `fscanf`. `RS232?` queries the RS-232 settings and returns the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

```
fprintf(s, 'RS232?')  
data = fscanf(s)  
data =  
9600;0;0;NONE;LF
```

Use the `textscan` function to parse and format the `data` variable into five new variables.

```
C = textscan(a, '%d%d%d%s%', 'delimiter', '');  
[br, sfc, hfc, par, tm] = deal(C{:});  
  
br =  
    9600  
sfc =  
    0  
hfc =  
    0  
par =  
    'NONE'  
tm =  
    'LF'
```

- 4 Disconnect and clean up — When you no longer need `s`, you should disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

```
fclose(s)  
delete(s)  
clear s
```

Example — Reading Binary Data

This example shows how you to download the TDS 210 oscilloscope screen display to MATLAB. The screen display data is transferred and saved to disk using the Windows bitmap format. This data provides a permanent record of your work, and is an easy way to document important signal and scope parameters.

Because the amount of data transferred is expected to be fairly large, it is asynchronously returned to the input buffer as soon as it is available from the instrument. This allows you to perform other tasks as the transfer progresses. Additionally, the scope is configured to its highest baud rate of 19,200.

- 1 Create a serial port object — Create the serial port object `s` associated with serial port COM1.

```
s = serial('COM1');
```

- 2 Configure property values — Configure the input buffer to accept a reasonably large number of bytes, and configure the baud rate to the highest value supported by the scope.

```
s.InputBufferSize = 50000;
s.BaudRate = 19200;
```

- 3 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

- 4 Write and read data — Configure the scope to transfer the screen display as a bitmap.

```
fprintf(s, 'HARDCOPY:PORT RS232')
fprintf(s, 'HARDCOPY:FORMAT BMP')
fprintf(s, 'HARDCOPY START')
```

Wait until all the data is sent to the input buffer, and then transfer the data to the MATLAB workspace as unsigned 8-bit integers.

```
out = fread(s,s.BytesAvailable,'uint8');
```

- 5 Disconnect and clean up — When you no longer need `s`, disconnect it from the instrument and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Viewing the Bitmap Data

To view the bitmap data, follow these steps:

- 1 Open a disk file.

- 2 Write the data to the disk file.
- 3 Close the disk file.
- 4 Read the data into MATLAB using the `imread` function.
- 5 Scale and display the data using the `imagesc` function.

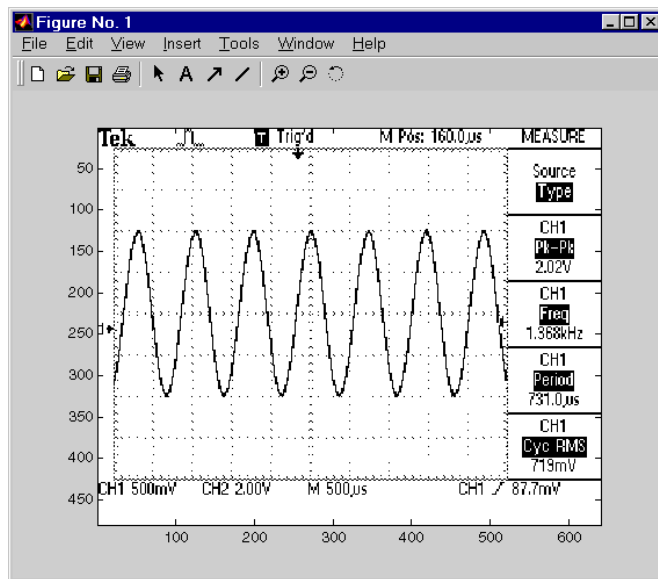
Note that the file I/O versions of the `fopen`, `fwrite`, and `fclose` functions are used.

```
fid = fopen('test1.bmp','w');
fwrite(fid,out,'uint8');
fclose(fid)
a = imread('test1.bmp','bmp');
imagesc(a)
```

Because the scope returns the screen display data using only two colors, an appropriate colormap is selected.

```
mymap = [0 0 0; 1 1 1];
colormap(mymap)
```

The following diagram shows the resulting bitmap image.



Events and Callbacks

In this section...

“Introduction” on page 23-91

“Example — Introduction to Events and Callbacks” on page 23-91

“Event Types and Callback Properties” on page 23-92

“Respond To Event Information” on page 23-94

“Create and Execute Callback Functions” on page 23-96

“Enable Callback Functions After They Error” on page 23-97

“Example — Using Events and Callbacks” on page 23-97

Introduction

You can enhance the power and flexibility of your serial port application by using events. An event occurs after a condition is met and might result in one or more callbacks.

While the serial port object is connected to the device, use events to display a message, display data, analyze data, and so on. Callbacks are controlled through callback properties and callback functions. All event types have an associated callback property. Callback functions are MATLAB functions that you construct to suit your specific application needs.

You execute a callback when a particular event occurs by specifying the name of the callback function as the value for the associated callback property.

Example — Introduction to Events and Callbacks

This example uses the callback function `instrcallback` to display a message to the command line when a bytes-available event occurs. The event is generated when the terminator is read.

```
s = serial('COM1');
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
fopen(s)
fprintf(s, '*IDN?')
out = fscanf(s);
```

MATLAB displays:

```
BytesAvailable event occurred at 17:01:29 for the object:
Serial-COM1.
```

End the serial port session.

```
fclose(s)
delete(s)
clear s
```

To see the code for the built-in `instrcallback` function, use the `type` command.

Event Types and Callback Properties

The following table describes serial port event types and callback properties. This table has two columns and nine rows. In the first column (event type), the second item (bytes available) applies to rows 2 through 4. Also, in the first column the last item (timer) applies to rows 8 and 9.

Event Types and Callback Properties

Event Type	Associated Properties
Break interrupt	BreakInterruptFcn
Bytes available	BytesAvailableFcn
	BytesAvailableFcnCount
	BytesAvailableFcnMode
Error	ErrorFcn
Output empty	OutputEmptyFcn
Pin status	PinStatusFcn
Timer	TimerFcn
	TimerPeriod

Break-Interrupt Event

A break-interrupt event is generated immediately after a break interrupt is generated by the serial port. The serial port generates a break interrupt when the received data has been in an inactive state longer than the transmission time for one character.

This event executes the callback function specified for the `BreakInterruptFcn` property. It can be generated for both synchronous and asynchronous read and write operations.

Bytes-Available Event

A bytes-available event is generated immediately after a predetermined number of bytes are available in the input buffer or a terminator is read, as determined by the `BytesAvailableFcnMode` property.

If `BytesAvailableFcnMode` is `byte`, the bytes-available event executes the callback function specified for the `BytesAvailableFcn` property every time the number of bytes specified by `BytesAvailableFcnCount` is stored in the input buffer. If `BytesAvailableFcnMode` is `terminator`, the callback function executes every time the character specified by the `Terminator` property is read.

This event can be generated only during an asynchronous read operation.

Error Event

An error event is generated immediately after an error occurs.

This event executes the callback function specified for the `ErrorFcn` property. It can be generated only during an asynchronous read or write operation.

An error event is generated when a time-out occurs. A time-out occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

Output-Empty Event

An output-empty event is generated immediately after the output buffer is empty.

This event executes the callback function specified for the `OutputEmptyFcn` property. It can be generated only during an asynchronous write operation.

Pin Status Event

A pin status event is generated immediately after the state (pin value) changes for the CD, CTS, DSR, or RI pins. For a description of these pins, see “Serial Port Signals and Pin Assignments” on page 23-55.

This event executes the callback function specified for the `PinStatusFcn` property. It can be generated for both synchronous and asynchronous read and write operations.

Timer Event

A timer event is generated when the time specified by the `TimerPeriod` property passes. Time is measured relative to when the serial port object is connected to the device.

This event executes the callback function specified for the `TimerFcn` property. Note that some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

Respond To Event Information

You can respond to event information in a callback function or in a record file. Event information is stored in a callback function using two fields: `Type` and `Data`. The `Type` field contains the event type, while the `Data` field contains event-specific information. As described in “Create and Execute Callback Functions” on page 23-96, these two fields are associated with a structure that you define in the callback function header. To learn about recording data and event information to a record file, see “Debugging: Recording Information to Disk” on page 23-105.

The following table shows event types and the values for the `Type` and `Data` fields. The table has three columns and 15 rows. Items in the first column (event type) span several rows, as follows:

Break interrupt: rows 1 and 2

Bytes available: rows 3 and 4

Error: rows 5 through 7

Output empty: rows 8 and 9

Pin status: rows 10 through 13

Timer: rows 14 and 15

Event Information

Event Type	Field	Field Value
Break interrupt	Type	BreakInterrupt
	Data.AbsTime	day-month-year hour:minute:second
Bytes available	Type	BytesAvailable
	Data.AbsTime	day-month-year hour:minute:second
Error	Type	Error
	Data.AbsTime	day-month-year hour:minute:second
	Data.Message	An error string
Output empty	Type	OutputEmpty
	Data.AbsTime	day-month-year hour:minute:second
Pin status	Type	PinStatus
	Data.AbsTime	day-month-year hour:minute:second
	Data.Pin	CarrierDetect, ClearToSend, DataSetReady, or RingIndicator
	Data.PinValue	on or off
Timer	Type	Timer
	Data.AbsTime	day-month-year hour:minute:second

The following topics describe the Data field values.

AbsTime Field

The AbsTime field, defined for all events, is the absolute time the event occurred. The absolute time is returned using the clock format: day-month-year hour:minute:second.

Pin Field

The pin status event uses the `Pin` field to indicate if the CD, CTS, DSR, or RI pins changed state. For a description of these pins, see “Serial Port Signals and Pin Assignments” on page 23-55.

PinValue Field

The pin status event uses the `PinValue` field to indicate the state of the CD, CTS, DSR, or RI pins. Possible values are `on` or `off`.

Message Field

The error event uses the `Message` field to store the descriptive message that is generated when an error occurs.

Create and Execute Callback Functions

To specify the callback function to be executed when a specific event type occurs, include the name of the file as the value for the associated callback property. You can specify the callback function as a function handle or as a string cell array element. For information about function handles, see “Create Function Handle”.

For example, to execute the callback function `mycallback` every time the terminator is read from your device:

```
s.BytesAvailableFcnMode = 'terminator';  
s.BytesAvailableFcn = @mycallback;
```

Alternatively, you can specify the callback function as a cell array.

```
s.BytesAvailableFcn = {'mycallback'};
```

Callback functions require at least two input arguments. The first argument is the serial port object. The second argument is a variable that captures the event information shown in the table, Event Information. This event information pertains only to the event that caused the callback function to execute. The function header for `mycallback` is:

```
function mycallback(obj,event)
```

You pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. For example, to pass the MATLAB variable `time` to `mycallback`:

```
time = datestr(now,0);  
s.BytesAvailableFcnMode = 'terminator';  
s.BytesAvailableFcn = {@mycallback,time};
```

Alternatively, you can specify the callback function as a string in the cell array.

```
s.BytesAvailableFcn = {'mycallback',time};
```

The corresponding function header is:

```
function mycallback(obj,event,time)
```

If you pass additional parameters to the callback function, they must be included in the function header after the two required arguments.

Note You can also specify the callback function as a string. In this case, the callback is evaluated in the MATLAB workspace and no requirements are made on the input arguments of the callback function.

Enable Callback Functions After They Error

If an error occurs while a callback function is executing the following occurs:

- The callback function is automatically disabled.
- A warning is displayed at the command line, indicating that the callback function is disabled.

If you want to enable the same callback function, set the callback property to the same value or disconnect the object with the `fclose` function. If you want to use a different callback function, the callback is enabled when you configure the callback property to the new value.

Example — Using Events and Callbacks

This example uses the callback function `instrcallback` to display event-related information to the command line when a bytes-available event or an output-empty event occurs.

- 1 Create a serial port object — Create the serial port object `s` associated with serial port COM1.

```
s = serial('COM1');
```

- 2 Configure properties — Configure `s` to execute the callback function `instrcallback` when a bytes-available event or an output-empty event occurs. Because `instrcallback` requires the serial port object and event information to be passed as input arguments, the callback function is specified as a function handle.

```
s.BytesAvailableFcnMode = 'terminator';  
s.BytesAvailableFcn = @instrcallback;  
s.OutputEmptyFcn = @instrcallback;
```

- 3 Connect to the device — Connect `s` to the Tektronix TDS 210 oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

- 4 Write and read data — Write the `RS232?` command asynchronously to the oscilloscope. This command queries the RS-232 settings and returns the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

```
fprintf(s, 'RS232?', 'async')
```

`instrcallback` is called after the `RS232?` command is sent, and when the terminator is read. The resulting displays are:

```
OutputEmpty event occurred at 17:37:21 for the object:  
Serial-COM1.
```

```
BytesAvailable event occurred at 17:37:21 for the object:  
Serial-COM1.
```

Read the data from the input buffer.

```
out = fscanf(s)  
out =  
9600;0;0;NONE;LF
```

- 5 Disconnect and clean up — When you no longer need `s`, disconnect it from the instrument and remove it from memory and from the MATLAB workspace.

```
fclose(s)  
delete(s)  
clear s
```


Control Pins

In this section...

“Properties of Serial Port Control Pins” on page 23-99
 “Signaling the Presence of Connected Devices” on page 23-99
 “Controlling the Flow of Data: Handshaking” on page 23-102

Properties of Serial Port Control Pins

As described in “Serial Port Signals and Pin Assignments” on page 23-55, 9-pin serial ports include six control pins. The following table shows properties associated with the serial port control pins.

Control Pin Properties

Property Name	Description
“DataTerminalReady”	State of the DTR pin
“FlowControl”	Data flow control method to use
“PinStatus”	State of the CD, CTS, DSR, and RI pins
“RequestToSend”	State of the RTS pin

Signaling the Presence of Connected Devices

DTEs and DCEs often use the CD, DSR, RI, and DTR pins to indicate whether a connection is established between serial port devices. Once the connection is established, you can begin to write or read data.

To monitor the state of the CD, DSR, and RI pins, use the `PinStatus` property. To specify or monitor the state of the DTR pin, use the `DataTerminalReady` property.

The following example illustrates how these pins are used when two modems are connected to each other.

Example — Connecting Two Modems

This example connects two modems to each other via the same computer, and illustrates how to monitor the communication status for the computer-modem connections, and for

the modem-modem connection. The first modem is connected to COM1, while the second modem is connected to COM2.

- 1 Create the serial port objects — After the modems are powered on, the serial port object `s1` is created for the first modem, and the serial port object `s2` is created for the second modem.

```
s1 = serial('COM1');  
s2 = serial('COM2');
```

- 2 Connect to the devices — `s1` and `s2` are connected to the modems. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffers as soon as it is available from the modems.

```
fopen(s1)  
fopen(s2)
```

Because the default `DataTerminalReady` property value is `on`, the computer (data terminal) is now ready to exchange data with the modems. To verify that the modems (data sets) can communicate with the computer, examine the value of the `Data Set Ready` pin using the `PinStatus` property.

```
s1.Pinstatus  
ans =  
    CarrierDetect: 'off'  
    ClearToSend: 'on'  
    DataSetReady: 'on'  
    RingIndicator: 'off'
```

The value of the `DataSetReady` field is `on` because both modems were powered on before they were connected to the objects.

- 3 Configure properties — Both modems are configured for a baud rate of 2400 bits per second and a carriage return (CR) terminator.

```
s1.BaudRate = 2400;  
s1.Terminator = 'CR';  
s2.BaudRate = 2400;  
s2.Terminator = 'CR';
```

- 4 Write and read data — Write the `atd` command to the first modem. This command puts the modem “off the hook,” which is equivalent to manually lifting a phone receiver.

```
fprintf(s1, 'atd')
```

Write the `ata` command to the second modem. This command puts the modem in “answer mode,” which forces it to connect to the first modem.

```
fprintf(s2, 'ata')
```

After the two modems negotiate their connection, verify the connection status by examining the value of the Carrier Detect pin using the `PinStatus` property.

```
s1.PinStatus
ans =
    CarrierDetect: 'on'
      ClearToSend: 'on'
    DataSetReady: 'on'
    RingIndicator: 'off'
```

Verify the modem-modem connection by reading the descriptive message returned by the second modem.

```
s2.BytesAvailable
ans =
    25
out = fread(s2,25);
char(out)'
ans =
ata
CONNECT 2400/NONE
```

Now break the connection between the two modems by configuring the `DataTerminalReady` property to `off`. To verify the modems are disconnected, examine the Carrier Detect pin value.

```
s1.DataTerminalReady = 'off';
s1.PinStatus
ans =
    CarrierDetect: 'off'
      ClearToSend: 'on'
    DataSetReady: 'on'
    RingIndicator: 'off'
```

- 5 Disconnect and clean up — Disconnect the objects from the modems and remove the objects from memory and from the MATLAB workspace.

```
fclose([s1 s2])
delete([s1 s2])
clear s1 s2
```

Controlling the Flow of Data: Handshaking

Data flow control or handshaking is a method used for communicating between a DCE and a DTE to prevent data loss during transmission. For example, suppose your computer can receive only a limited amount of data before it must be processed. As this limit is reached, a handshaking signal is transmitted to the DCE to stop sending data. When the computer can accept more data, another handshaking signal is transmitted to the DCE to resume sending data.

If supported by your device, you can control data flow using one of these methods:

- Hardware handshaking on page 23-102
- Software handshaking on page 23-103

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

To specify the data flow control method, use the `FlowControl` property. If `FlowControl` is `hardware`, hardware handshaking is used to control data flow. If `FlowControl` is `software`, software handshaking is used to control data flow. If `FlowControl` is `none`, no handshaking is used.

Hardware Handshaking

Hardware handshaking uses specific serial port pins to control data flow. In most cases, these are the RTS and CTS pins. Hardware handshaking using these pins is described in “The RTS and CTS Pins” on page 23-58.

If `FlowControl` is `hardware`, the RTS and CTS pins are automatically managed by the DTE and DCE. To return the CTS pin value, use the `PinStatus` property. Configure or return the RTS pin value with the `RequestToSend` property.

Note Some devices also use the DTR and DSR pins for handshaking. However, these pins are typically used to indicate that the system is ready for communication, and are not used to control data transmission. In MATLAB, hardware handshaking always uses the RTS and CTS pins.

If your device does not use hardware handshaking in the standard way, then you might need to manually configure the `RequestToSend` property. In this case, you should configure `FlowControl` to `none`. If `FlowControl` is hardware, then the `RequestToSend` value that you specify might not be honored. Refer to the device documentation to determine its specific pin behavior.

Software Handshaking

Software handshaking uses specific ASCII characters to control data flow. These characters, known as Xon and Xoff (or XON and XOFF), are described in the following table.

Software Handshaking Characters

Character	Integer Value	Description
Xon	17	Resume data transmission
Xoff	19	Pause data transmission

When using software handshaking, the control characters are sent over the transmission line the same way as regular data. Therefore, only the TD, RD, and GND pins are needed.

The main disadvantage of software handshaking is that Xon or Xoff characters are not writable while numerical data is being written to the device. This is because numerical data might contain a 17 or 19, which makes it impossible to distinguish between the control characters and the data. However, you can write Xon or Xoff while data is being asynchronously read from the device because you are using both the TD and RD pins.

Example: Using Software Handshaking

Suppose you want to use software flow control with the example described in “Example — Reading Binary Data” on page 23-88. To do this, you must configure the oscilloscope and serial port object for software flow control.

```
fprintf(s, 'RS232:SOFTF ON')
s.FlowControl = 'software';
```

To pause data transfer, write the numerical value 19 to the device.

```
fwrite(s, 19)
```

To resume data transfer, write the numerical value 17 to the device.

`fwrite(s,17)`

Debugging: Recording Information to Disk

In this section...

“Introduction” on page 23-105

“Recording Properties” on page 23-105

“Example: Introduction to Recording Information” on page 23-106

“Creating Multiple Record Files” on page 23-106

“Specifying a Filename” on page 23-107

“The Record File Format” on page 23-107

“Example: Recording Information to Disk” on page 23-108

Introduction

Recording information to disk provides a permanent record of your serial port session, and is an easy way to debug your application. While the serial port object is connected to the device, you can record the following information to a disk file:

- The number of values written to the device, the number of values read from the device, and the data type of the values
- Data written to the device, and data read from the device
- Event information

Recording Properties

You record information to a disk file with the `record` function. The following table shows the properties associated with recording information to disk.

Recording Properties

Property Name	Description
"RecordDetail"	Amount of information saved to a record file
"RecordMode"	Specify whether data and event information is saved to one record file or to multiple record files
"RecordName"	Name of the record file
"RecordStatus"	Indicate if data and event information are saved to a record file

Example: Introduction to Recording Information

This example records the number of values written to and read from the device, and stores the information to the file `myfile.txt`.

```
s = serial('COM1');
fopen(s)
s.RecordName = 'myfile.txt';
record(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fprintf(s, 'RS232?')
rs232 = fscanf(s);
```

End the serial port session.

```
fclose(s)
delete(s)
clear s
```

To display `myfile.txt` at the command line, use the `type` command.

Creating Multiple Record Files

When you initiate recording with the `record` function, the `RecordMode` property determines if a new record file is created or if new information is appended to an existing record file.

You can configure `RecordMode` to `overwrite`, `append`, or `index`. If `RecordMode` is `overwrite`, the record file is overwritten each time recording is initiated. If `RecordMode` is `append`, the new information is appended to the file specified by `RecordMode`. If

`RecordMode` is `index`, a different disk file is created each time recording is initiated. The rules for specifying a record filename are discussed in the next section.

Specifying a Filename

You specify the name of the record file with the `RecordName` property. You can specify any value for `RecordName` — including a directory path — provided the filename is supported by your operating system. Additionally, if `RecordMode` is `index`, the filename follows these rules:

- Indexed filenames are identified by a number. This number precedes the filename extension and is increased by 1 for successive record files.
- If no number is specified as part of the initial filename, the first record file does not have a number associated with it. For example, if `RecordName` is `myfile.txt`, `myfile.txt` is the name of the first record file, `myfile01.txt` is the name of the second record file, and so on.
- `RecordName` is updated after the record file is closed.
- If the specified filename already exists, the existing file is overwritten.

The Record File Format

The record file is an ASCII file that contains a record of one or more serial port sessions. You specify the amount of information saved to a record file with the `RecordDetail` property.

`RecordDetail` can be `compact` or `verbose`. A compact record file contains the number of values written to the device, the number of values read from the device, the data type of the values, and event information. A verbose record file contains the preceding information as well as the data transferred to and from the device.

Binary data with precision given by `uchar`, `schar`, `(u)int8`, `(u)int16`, or `(u)int32` is recorded using hexadecimal format. For example, if the integer value 255 is read from the instrument as a 16-bit integer, the hexadecimal value 00FF is saved in the record file. Single- and double-precision floating-point numbers are recorded as decimal values using the `%g` format, and as hexadecimal values using the format specified by the IEEE® Standard 754-1985 for Binary Floating-Point Arithmetic.

The IEEE floating-point format includes three components: the sign bit, the exponent field, and the significant field. Single-precision floating-point values consist of 32 bits. The value is given by

$$\text{value} = (-1)^{\text{sign}} (2^{\text{exp}-127}) (1.\text{significand})$$

Double-precision floating-point values consist of 64 bits. The value is given by

$$\text{value} = (-1)^{\text{sign}} (2^{\text{exp}-1023}) (1.\text{significand})$$

The floating-point format component, and the associated single-precision and double-precision bits are shown in the following table.

Component	Single-Precision Bits	Double-Precision Bits
sign	1	1
exp	2-9	2-12
significand	10-32	13-64

Bit 1 is the left-most bit as stored in the record file.

Example: Recording Information to Disk

This example illustrates how to record information transferred between a serial port object and a Tektronix TDS 210 oscilloscope. Additionally, the structure of the resulting record file is presented.

- 1 Create the serial port object — Create the serial port object `s` associated with the serial port COM1.

```
s = serial('COM1');
```

- 2 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

- 3 Configure property values — Configure `s` to record information to multiple disk files using the verbose format. Recording is then initiated with the first disk file defined as `WaveForm1.txt`.

```
s.RecordMode = 'index';
s.RecordDetail = 'verbose';
s.RecordName = 'WaveForm1.txt';
record(s)
```

- 4 Write and read data — The commands written to the instrument, and the data read from the instrument are recorded in the record file. For an explanation of the oscilloscope commands, see “Example — Writing and Reading Text Data” on page 23-86.

```
fprintf(s,'*IDN?')
idn = fscanf(s);
fprintf(s,'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s);
```

Read the peak-to-peak voltage with the `fread` function. Note that the data returned by `fread` is recorded using hex format.

```
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
ptop = fread(s,s.BytesAvailable);
```

Convert the peak-to-peak voltage to a character array.

```
char(ptop)'
ans =
2.0199999809E0
```

The recording state is toggled from on to off. Because the `RecordMode` value is `index`, the record filename is automatically updated.

```
record(s)
s.RecordStatus
ans =
off
s.RecordName
ans =
WaveForm2.txt
```

- 5 Disconnect and clean up — When you no longer need `s`, disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

The Record File Contents

The contents of the WaveForm1.txt record file are shown below. Because the RecordDetail property was verbose, the number of values, commands, and data were recorded. Note that data returned by the fread function is in hex format.

```
type WaveForm1.txt
```

```
Legend:
```

```
* - An event occurred.  
> - A write operation occurred.  
< - A read operation occurred.  
1   Recording on 22-Jan-2000 at 11:21:21.575. Binary data in...  
2   > 6 ascii values.  
    *IDN?  
3   < 56 ascii values.  
    TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04  
4   > 29 ascii values.  
    MEASUREMENT:IMMED:SOURCE CH2  
5   > 26 ascii values.  
    MEASUREMENT:IMMED:SOURCE?  
6   < 4 ascii values.  
    CH2  
7   > 27 ascii values.  
    MEASUREMENT:MEAS1:TYPE PK2PK  
8   > 25 ascii values.  
    MEASUREMENT:MEAS1:VALUE?  
9   < 15 uchar values.  
    32 2e 30 31 39 39 39 39 39 38 30 39 45 30 0a  
10  Recording off.
```

Save and Load

In this section...

“Using save and load” on page 23-111

“Using Serial Port Objects on Different Platforms” on page 23-111

Using save and load

Save serial port objects to a file, just as you would any workspace variable, using the `save` command. For example, suppose you create the serial port object `s` associated with the serial port COM1, configure several property values, and perform a write and read operation.

```
s = serial('COM1');  
s.BaudRate = 19200;  
s.Tag = 'My serial object';  
fopen(s)  
fprintf(s, '*IDN?')  
out = fscanf(s);
```

To save the serial port object and the data read from the device to the file `myserial.mat`:

```
save myserial s out
```

Note To save data and event information as text to a disk file, use the `record` function.

You can recreate `s` and `out` in the workspace using the `load` command.

```
load myserial
```

Values for read only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. Therefore, to use `s`, you must connect it to the device with the `fopen` function. To determine if a property is read only, examine its reference pages.

Using Serial Port Objects on Different Platforms

If you save a serial port object from one platform, and then load that object on a different platform having different serial port names, you need to modify the `Port` property value.

For example, suppose you create the serial port object `s` associated with the serial port COM1 on a Microsoft Windows platform. If you want to save `s` for eventual use on a Linux platform, configure `Port` to an appropriate value such as `ttyS0` after the object is loaded.

Disconnect and Clean Up

In this section...

“Disconnect a Serial Port Object” on page 23-113

“Clean Up the MATLAB Environment” on page 23-113

Disconnect a Serial Port Object

When you no longer need to communicate with the device, disconnect it from the serial port object with the `fclose` function.

```
fclose(s)
```

Examine the `Status` property to verify that the serial port object and the device are disconnected.

```
s.Status  
ans =  
closed
```

After `fclose` is issued, the serial port associated with `s` is available. Now connect another serial port object to it using `fopen`.

Clean Up the MATLAB Environment

When the serial port object is no longer needed, remove it from memory with the `delete` function.

```
delete(s)
```

Before using `delete`, disconnect the serial port object from the device with the `fclose` function.

A deleted serial port object is invalid, which means that you cannot connect it to the device. In this case, remove the object from the MATLAB workspace. To remove serial port objects and other variables from the MATLAB workspace, use the `clear` command.

```
clear s
```

Use `clear` on a serial port object that is still connected to a device to remove the object from the workspace but leave it connected to the device. Restore cleared objects to MATLAB with the `instrfind` function.

Property Reference

In this section...

“Configure Serial Port Properties” on page 23-115

“Serial Port Object Properties” on page 23-115

Configure Serial Port Properties

Properties can be set after you create the `serial` object. A read-only property is called as follows.

```
s = serial('COM1');
s.TransferStatus
ans =
    idle
```

A property you can configure is set as follows.

```
s = serial('COM1');
s.Timeout = 30;
```

Serial Port Object Properties

The serial port object properties are briefly described below, and organized into categories based on how they are used.

Communications Properties	
“BaudRate”	Rate at which bits are transmitted
“DataBits”	Number of data bits to transmit
“Parity”	Type of parity checking
“StopBits”	Number of bits used to indicate the end of a byte
“Terminator”	Terminator character
Write Properties	
“BytesToOutput”	Number of bytes currently in the output buffer

Write Properties	
"OutputBufferSize"	Size of the output buffer in bytes
"Timeout"	Waiting time to complete a read or write operation
"TransferStatus"	Indicate if an asynchronous read or write operation is in progress
"ValuesSent"	Total number of values written to the device

Read Properties	
"BytesAvailable"	Number of bytes available in the input buffer
"InputBufferSize"	Size of the input buffer in bytes
"ReadAsyncMode"	Specify whether an asynchronous read operation is continuous or manual
"Timeout"	Waiting time to complete a read or write operation
"TransferStatus"	Indicate if an asynchronous read or write operation is in progress
"ValuesReceived"	Total number of values read from the device

Callback Properties	
BreakInterruptFcn	Callback function to execute when a break-interrupt event occurs
BytesAvailableFcn	Callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read
BytesAvailableFcnCount	Number of bytes that must be available in the input buffer to generate a bytes-available event
BytesAvailableFcnMode	Specify if the bytes-available event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read
ErrorFcn	Callback function to execute when an error event occurs
OutputEmptyFcn	Callback function to execute when the output buffer is empty
PinStatusFcn	Callback function to execute when the CD, CTS, DSR, or RI pins change state

Callback Properties	
TimerFcn	Callback function to execute when a predefined period of time passes
TimerPeriod	Period of time between timer events

Control Pin Properties	
"DataTerminalReady"	State of the DTR pin
"FlowControl"	Data flow control method to use
"PinStatus"	State of the CD, CTS, DSR, and RI pins
"RequestToSend"	State of the RTS pin

Recording Properties	
"RecordDetail"	Amount of information saved to a record file
"RecordMode"	Specify whether data and event information are saved to one record file or to multiple record files
"RecordName"	Name of the record file
"RecordStatus"	Indicate if data and event information are saved to a record file

General Purpose Properties	
"ByteOrder"	Order in which the device stores bytes
Name	Descriptive name for the serial port object
"port"	Platform-specific serial port name
"Status"	Indicate if the serial port object is connected to the device
Tag	Label to associate with a serial port object
Type	Object type
UserData	Data you want to associate with a serial port object

Properties — Alphabetical List

BaudRate

Specify the rate at which bits are transmitted

Description

You configure `BaudRate` as bits per second. The transferred bits include the start bit, the data bits, the parity bit (if used), and the stop bits. However, only the data bits are stored.

The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, “9600 baud” means that the serial port is capable of transferring a maximum of 9600 bits per second. If the information unit is one baud (one bit), the bit rate and the baud rate are identical. If one baud is given as 10 bits, (for example, eight data bits plus two framing bits), the bit rate is still 9600 but the baud rate is 9600/10, or 960. You always configure `BaudRate` as bits per second. Therefore, in the previous example, set `BaudRate` to 9600.

Note Both the computer and the peripheral device must be configured to the same baud rate before you can successfully read or write data.

Standard baud rates include 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 bits per second. To display the supported baud rates for the serial ports on your platform, see “Finding Serial Port Information for Your Platform” on page 23-62.

Characteristics

Read only	Never
Data type	Double

Values

The default value is 9600.

BreakInterruptFcn

Specify the callback function to execute when a break-interrupt event occurs

Description

You configure `BreakInterruptFcn` to execute a callback function when a break-interrupt event occurs. A break-interrupt event is generated by the serial port when the received data is in an off (space) state longer than the transmission time for one byte.

Note A break-interrupt event can be generated at any time during the serial port session.

If the `RecordStatus` property value is on, and a break-interrupt event occurs, the record file records this information:

- The event type as `BreakInterrupt`
- The time the event occurred using the format day-month-year
hour:minute:second:millisecond

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 23-96.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

ByteOrder

Specify the byte order of the device

Description

You configure `ByteOrder` to be `littleEndian` or `bigEndian`. If `ByteOrder` is `littleEndian`, the device stores the first byte in the first memory address. If `ByteOrder` is `bigEndian`, the device stores the last byte in the first memory address.

For example, suppose the hexadecimal value `4F52` is to be stored in device memory. Because this value consists of two bytes, `4F` and `52`, two memory locations are used. Using big-endian format, `4F` is stored first in the lower storage address. Using little-endian format, `52` is stored first in the lower storage address.

Note Configure `ByteOrder` to the appropriate value for your device before performing a read or write operation. Refer to your device documentation for information about the order in which it stores bytes.

Characteristics

Read only	Never
Data type	String

Values

<code>{littleEndian}</code>	The byte order of the device is little-endian.
<code>bigEndian</code>	The byte order of the device is big-endian.

BytesAvailable

Number of bytes available in the input buffer

Description

BytesAvailable indicates the number of bytes currently available to be read from the input buffer. The property value is continuously updated as the input buffer is filled, and is set to 0 after the fopen function is issued.

You can make use of BytesAvailable only when reading data asynchronously. This is because when reading data synchronously, control is returned to the MATLAB command line only after the input buffer is empty. Therefore, the BytesAvailable value is always 0. To learn how to read data asynchronously, see “Reading Text Data” on page 23-83.

The BytesAvailable value can range from zero to the size of the input buffer. Use the InputBufferSize property to specify the size of the input buffer. Use the ValuesReceived property to return the total number of values read.

Characteristics

Read only	Always
Data type	Double

Values

The default value is 0.

BytesAvailableFcn

Specify the callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read

Description

You configure `BytesAvailableFcn` to execute a callback function when a bytes-available event occurs. A bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available in the input buffer, or after a terminator is read, as determined by the `BytesAvailableFcnMode` property.

Note A bytes-available event can be generated only for asynchronous read operations.

If the `RecordStatus` property value is on, and a bytes-available event occurs, the record file records this information:

- The event type as `BytesAvailable`
- The time the event occurred using the format day-month-year
hour:minute:second:millisecond

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 23-96.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

Example

Create the serial port object `s` for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

```
s = serial('COM1');
```

Configure `s` to execute the callback function `instrcallback` when 40 bytes are available in the input buffer.

```
s.BytesAvailableFcnCount = 40;  
s.BytesAvailableFcnMode = 'byte';  
s.BytesAvailableFcn = @instrcallback;
```

Connect `s` to the oscilloscope.

```
fopen(s)
```

Write the `*IDN?` command, which instructs the scope to return identification information. Because the default value for the `ReadAsyncMode` property is `continuous`, data is read as soon as it is available from the instrument.

```
fprintf(s, '*IDN?')
```

MATLAB displays:

```
BytesAvailable event occurred at 18:33:35 for the object:  
Serial-COM1.
```

56 bytes are read and `instrcallback` is called once. The resulting display is shown above.

```
s.BytesAvailable  
ans =  
    56
```

Suppose you remove 25 bytes from the input buffer and then issue the `MEASUREMENT?` command, which instructs the scope to return its measurement settings.

```
out = fscanf(s, '%c', 25);  
fprintf(s, 'MEASUREMENT?')
```

MATLAB displays:

```
BytesAvailable event occurred at 18:33:48 for the object:  
Serial-COM1.
```

```
BytesAvailable event occurred at 18:33:48 for the object:  
Serial-COM1.
```

There are now 102 bytes in the input buffer, 31 of which are left over from the *IDN? command. `instrcallback` is called twice—once when 40 bytes are available and once when 80 bytes are available.

```
s.BytesAvailable  
ans =  
    102
```

BytesAvailableFcnCount

Specify the number of bytes that must be available in the input buffer to generate a bytes-available event

Description

You configure `BytesAvailableFcnCount` to the number of bytes that must be available in the input buffer before a bytes-available event is generated.

Use the `BytesAvailableFcnMode` property to specify whether the bytes-available event occurs after a certain number of bytes are available or after a terminator is read.

The bytes-available event executes the callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnCount` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	Double

Values

The default value is 48.

BytesAvailableFcnMode

Specify if the bytes-available event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read

Description

You can configure `BytesAvailableFcnMode` to be `terminator` or `byte`. If `BytesAvailableFcnMode` is `terminator`, a bytes-available event occurs when the terminator specified by the `Terminator` property is reached. If `BytesAvailableFcnMode` is `byte`, a bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available.

The bytes-available event executes the callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnMode` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	String

Values

<code>{terminator}</code>	A bytes-available event is generated when the terminator is read.
<code>byte</code>	A bytes-available event is generated when the specified number of bytes are available.

BytesToOutput

Number of bytes currently in the output buffer

Description

BytesToOutput indicates the number of bytes currently in the output buffer waiting to be written to the device. The property value is continuously updated as the output buffer is filled and emptied, and is set to 0 after the `fopen` function is issued.

You can make use of BytesToOutput only when writing data asynchronously. This is because when writing data synchronously, control is returned to the MATLAB command line only after the output buffer is empty. Therefore, the BytesToOutput value is always 0. To learn how to write data asynchronously, see “Writing Text Data” on page 23-77.

Use the ValuesSent property to return the total number of values written to the device.

Note If you attempt to write out more data than can fit in the output buffer, an error is returned and BytesToOutput is 0. Specify the size of the output buffer with the OutputBufferSize property.

Characteristics

Read only	Always
Data type	Double

Values

The default value is 0.

DataBits

Number of data bits to transmit

Description

You can configure `DataBits` to be 5, 6, 7, or 8. Data is transmitted as a series of five, six, seven, or eight bits with the least significant bit sent first. At least seven data bits are required to transmit ASCII characters. Eight bits are required to transmit binary data. Five and six bit data formats are used for specialized communications equipment.

Note Both the computer and the peripheral device must be configured to transmit the same number of data bits.

In addition to the data bits, the serial data format consists of a start bit, one or two stop bits, and possibly a parity bit. You specify the number of stop bits with the `StopBits` property, and the type of parity checking with the `Parity` property.

To display the supported number of data bits for the serial ports on your platform, see “Finding Serial Port Information for Your Platform” on page 23-62.

Characteristics

Read only	Never
Data type	Double

Values

`DataBits` can be 5, 6, 7, or 8. The default value is 8.

DataTerminalReady

State of the DTR pin

Description

You can configure `DataTerminalReady` to be `on` or `off`. If `DataTerminalReady` is `on`, the Data Terminal Ready (DTR) pin is asserted. If `DataTerminalReady` is `off`, the DTR pin is unasserted.

In normal usage, the DTR and Data Set Ready (DSR) pins work together, and are used to signal if devices are connected and powered. However, there is nothing in the RS-232 standard that states the DTR pin must be used in any specific way. For example, DTR and DSR might be used for handshaking. You should refer to your device documentation to determine its specific pin behavior.

You can return the value of the DSR pin with the `PinStatus` property. Handshaking is described in “Controlling the Flow of Data: Handshaking” on page 23-102.

Characteristics

Read only	Never
Data type	String

Values

<code>{on}</code>	The DTR pin is asserted.
<code>off</code>	The DTR pin is unasserted.

ErrorFcn

Specify the callback function to execute when an error event occurs

Description

You configure `ErrorFcn` to execute a callback function when an error event occurs.

Note An error event is generated only for asynchronous read and write operations.

An error event is generated when a time-out occurs. A time-out occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

If the `RecordStatus` property value is on, and an error event occurs, the record file records this information:

- The event type as `Error`
- The error message
- The time the event occurred using the format `day-month-year hour:minute:second:millisecond`

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 23-96.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

FlowControl

Data flow control method to use

Description

You can configure `FlowControl` to be `none`, `hardware`, or `software`. If `FlowControl` is `none`, data flow control (handshaking) is not used. If `FlowControl` is `hardware`, hardware handshaking is used to control data flow. If `FlowControl` is `software`, software handshaking is used to control data flow.

Hardware handshaking typically utilizes the Request to Send (RTS) and Clear to Send (CTS) pins to control data flow. Software handshaking uses control characters (Xon and Xoff) to control data flow. For more information about handshaking, see “Controlling the Flow of Data: Handshaking” on page 23-102.

You can return the value of the CTS pin with the `PinStatus` property. You can specify the value of the RTS pin with the `RequestToSend` property. However, if `FlowControl` is `hardware`, and you specify a value for `RequestToSend`, that value might not be honored.

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

Characteristics

Read only	Never
Data type	String

Values

{none}	No flow control is used.
--------	--------------------------

hardware	Hardware flow control is used.
software	Software flow control is used.

InputBufferSize

Size of the input buffer in bytes

Description

You configure `InputBufferSize` as the total number of bytes that can be stored in the input buffer during a read operation.

A read operation is terminated if the amount of data stored in the input buffer equals the `InputBufferSize` value. You can read text data with the `fgetl`, `fget`, or `fscanf` functions. You can read binary data with the `fread` function.

You can configure `InputBufferSize` only when the serial port object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

If you configure `InputBufferSize` while there is data in the input buffer, that data is flushed.

Characteristics

Read only	While open
Data type	Double

Values

The default value is 512.

Name

Descriptive name for the serial port object

Description

You configure `Name` to be a descriptive name for the serial port object.

When you create a serial port object, a descriptive name is automatically generated and stored in `Name`. This name is given by concatenating the word “Serial” with the serial port specified in the `serial` function. However, you can change the value of `Name` at any time.

The serial port is given by the `Port` property. If you modify this property value, then `Name` is automatically updated to reflect that change.

Characteristics

Read only	Never
Data type	String

Values

`Name` is automatically defined when the serial port object is created.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
```

`s` is automatically assigned a descriptive name.

```
s.Name  
ans =  
Serial-COM1
```


ObjectVisibility

Control access to serial port object

Description

The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the serial port objects created by their applications. When an object's `ObjectVisibility` property is set to `off`, `instrfind` does not return or delete that object.

Objects that are not visible are still valid. If you have access to the object (for example, from within the file that creates it), you can set and get its properties and pass it to any function that operates on serial port objects.

Characteristics

Usage	Any serial port object
Read only	Never
Data type	String

Values

<code>{on}</code>	Object is visible to <code>instrfind</code> .
<code>off</code>	Object is not visible from the command line (except by <code>instrfindall</code>).

Examples

The following statement creates a serial port object with its `ObjectVisibility` property set to `off`:

```
s = serial('COM1','ObjectVisibility','off');
instrfind
ans =
     []
```

However, because the hidden object is in the workspace (s), you can access it.

```
get(s,'ObjectVisibility')
ans =
    off
```

See Also

Functions

`instrfind`, `instrfindall`

OutputBufferSize

Size of the output buffer in bytes

Description

You configure `OutputBufferSize` as the total number of bytes that can be stored in the output buffer during a write operation.

An error occurs if the output buffer cannot hold all the data to be written. You write text data with the `fprintf` function. You write binary data with the `fwrite` function.

You can configure `OutputBufferSize` only when the serial port object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	Double

Values

The default value is 512.

OutputEmptyFcn

Specify the callback function to execute when the output buffer is empty

Description

You configure `OutputEmptyFcn` to execute a callback function when an output-empty event occurs. An output-empty event is generated when the last byte is sent from the output buffer to the device.

Note An output-empty event can be generated only for asynchronous write operations.

If the `RecordStatus` property value is on, and an output-empty event occurs, the record file records this information:

- The event type as `OutputEmpty`
- The time the event occurred using the format day-month-year
hour:minute:second:millisecond

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 23-96.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

Parity

Specify the type of parity checking

Description

You can configure **Parity** to be **none**, **odd**, **even**, **mark**, or **space**. If **Parity** is **none**, parity checking is not performed and the parity bit is not transmitted. If **Parity** is **odd**, the number of mark bits (1s) in the data is counted, and the parity bit is asserted or unasserted to obtain an odd number of mark bits. If **Parity** is **even**, the number of mark bits in the data is counted, and the parity bit is asserted or unasserted to obtain an even number of mark bits. If **Parity** is **mark**, the parity bit is asserted. If **Parity** is **space**, the parity bit is unasserted.

Parity checking can detect errors of one bit only. An error in two bits might cause the data to have a seemingly valid parity, when in fact it is incorrect. For more information about parity checking, see “The Parity Bit” on page 23-61.

In addition to the parity bit, the serial data format consists of a start bit, between five and eight data bits, and one or two stop bits. You specify the number of data bits with the **DataBits** property, and the number of stop bits with the **StopBits** property.

Characteristics

Read only	Never
Data type	String

Values

{none}	No parity checking
odd	Odd parity checking
even	Even parity checking

mark	Mark parity checking
space	Space parity checking

PinStatus

State of the CD, CTS, DSR, and RI pins

Description

`PinStatus` is a structure array that contains the fields `CarrierDetect`, `ClearToSend`, `DataSetReady` and `RingIndicator`. These fields indicate the state of the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) and Ring Indicator (RI) pins, respectively. For more information about these pins, see “Serial Port Signals and Pin Assignments” on page 23-55.

`PinStatus` can be `on` or `off` for any of these fields. A value of `on` indicates the associated pin is asserted. A value of `off` indicates the associated pin is unasserted. A pin status event occurs when any of these pins changes its state. A pin status event executes the call back function specified by `PinStatusFcn`.

In normal usage, the Data Terminal Ready (DTR) and DSR pins work together, while the Request to Send (RTS) and CTS pins work together. You can specify the state of the DTR pin with the `DataTerminalReady` property. You can specify the state of the RTS pin with the `RequestToSend` property.

For an example that uses `PinStatus`, see “Example — Connecting Two Modems” on page 23-99.

Characteristics

Read only	Always
Data type	Structure

Values

<code>off</code>	The associated pin is unasserted.
------------------	-----------------------------------

on	The associated pin is asserted.
----	---------------------------------

The default value is device dependent.

PinStatusFcn

Specify the callback function to execute when the CD, CTS, DSR, or RI pins change state

Description

You configure `PinStatusFcn` to execute a callback function when a pin status event occurs. A pin status event occurs when the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) or Ring Indicator (RI) pin changes state. A serial port pin changes state when it is asserted or unasserted. Information about the state of these pins is recorded in the `PinStatus` property.

Note A pin status event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a pin status event occurs, the record file records this information:

- The event type as `PinStatus`
- The pin that changed its state, and the pin state as either `on` or `off`
- The time the event occurred using the format `day-month-year hour:minute:second:millisecond`

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 23-96.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

Port

Specify the platform-specific serial port name

Description

You configure `Port` to be the name of a serial port on your platform. `Port` specifies the physical port associated with the object and the device.

When you create a serial port object, `Port` is automatically assigned the port name specified for the `serial` function.

You can configure `Port` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	String

Values

The `Port` value is determined when the serial port object is created.

Example

Suppose you create a serial port object associated with serial port `COM1`.

```
s = serial('COM1');
```

The value of the `Port` property is `COM1`.

```
s.Port  
ans =  
COM1
```

ReadAsyncMode

Specify whether an asynchronous read operation is continuous or manual

Description

You can configure `ReadAsyncMode` to be `continuous` or `manual`. If `ReadAsyncMode` is `continuous`, the serial port object continuously queries the device to determine if data is available to be read. If data is available, it is automatically read and stored in the input buffer. If issued, the `readasync` function is ignored.

If `ReadAsyncMode` is `manual`, the object does not query the device to determine if data is available to be read. Instead, you must manually issue the `readasync` function to perform an asynchronous read operation. Because `readasync` checks for the terminator, this function can be slow. To increase speed, configure `ReadAsyncMode` to `continuous`.

Note If the device is ready to transmit data, it will do so regardless of the `ReadAsyncMode` value. Therefore, if `ReadAsyncMode` is `manual` and a read operation is not in progress, data might be lost. To guarantee that all transmitted data is stored in the input buffer, you should configure `ReadAsyncMode` to `continuous`.

To determine the amount of data available in the input buffer, use the `BytesAvailable` property. For either `ReadAsyncMode` value, you can bring data into the MATLAB workspace with one of the synchronous read functions such as `fscanf`, `fgetl`, `fgets`, or `fread`.

Characteristics

Read only	Never
Data type	String

Values

<code>{continuous}</code>	Continuously query the device to determine if data is available to be read.
<code>manual</code>	Manually read data from the device using the <code>readasync</code> function.

RecordDetail

Specify the amount of information saved to a record file

Description

You can configure `RecordDetail` to be `compact` or `verbose`. If `RecordDetail` is `compact`, the number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file. If `RecordDetail` is `verbose`, the data written to the device, and the data read from the device are also saved to the record file.

The event information saved to a record file is shown in the table, Event Information. The verbose record file structure is shown in “Example: Recording Information to Disk” on page 23-108.

Characteristics

Read only	Never
Data type	String

Values

<code>{compact}</code>	The number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file.
<code>verbose</code>	The data written to the device, and the data read from the device are also saved to the record file.

RecordMode

Specify whether data and event information are saved to one record file or to multiple record files

Description

You can configure `RecordMode` to be `overwrite`, `append`, or `index`. If `RecordMode` is `overwrite`, the record file is overwritten each time recording is initiated. If `RecordMode` is `append`, data is appended to the record file each time recording is initiated. If `RecordMode` is `index`, a different record file is created each time recording is initiated, each with an indexed filename.

You can configure `RecordMode` only when the object is not recording. You terminate recording with the `record` function. A object that is not recording has a `RecordStatus` property value of `off`.

You specify the record filename with the `RecordName` property. The indexed filename follows a prescribed set of rules. For a description of these rules, see “Specifying a Filename” on page 23-107.

Characteristics

Read only	While recording
Data type	String

Values

<code>{overwrite}</code>	The record file is overwritten.
<code>append</code>	Data is appended to an existing record file.
<code>index</code>	A different record file is created, each with an indexed filename.

Example

Suppose you create the serial port object `s` associated with the serial port COM1.

```
s = serial('COM1');  
fopen(s)
```

Specify the record filename with the `RecordName` property, configure `RecordMode` to `index`, and initiate recording.

```
s.RecordName = 'MyRecord.txt';  
s.RecordMode = 'index';  
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```
record(s, 'off')  
s.RecordName  
ans =  
MyRecord01.txt
```

Disconnect `s` from the peripheral device, remove `s` from memory, and remove `s` from the MATLAB workspace.

```
fclose(s)  
delete(s)  
clear s
```

RecordName

Name of the record file

Description

You configure `RecordName` to be the name of the record file. You can specify any value for `RecordName` - including a directory path - provided the file name is supported by your operating system.

MATLAB software supports any file name supported by your operating system. However, if you access the file with a MATLAB command, you might need to specify the file name using single quotes. For example, suppose you name the record file `My Record.txt`. To type this file at the MATLAB command line, you must include the name in quotes.

```
type('My Record.txt')
```

You can specify whether data and event information are saved to one disk file or to multiple disk files with the `RecordMode` property. If `RecordMode` is `index`, the filename follows a prescribed set of rules. For a description of these rules, see “Specifying a Filename” on page 23-107.

You can configure `RecordName` only when the object is not recording. You terminate recording with the `record` function. An object that is not recording has a `RecordStatus` property value of `off`.

Characteristics

Read only	While recording
Data type	String

Values

The default record filename is `record.txt`.

RecordStatus

Indicate if data and event information are saved to a record file

Description

You can configure RecordStatus to be off or on with the record function. If RecordStatus is off, then data and event information are not saved to a record file. If RecordStatus is on, then data and event information are saved to the record file specified by RecordName.

Use the record function to initiate or complete recording. RecordStatus is automatically configured to reflect the recording state.

For more information about recording to a disk file, see “Debugging: Recording Information to Disk” on page 23-105.

Characteristics

Read only	Always
Data type	String

Values

{off}	Data and event information are not written to a record file.
on	Data and event information are written to a record file.

RequestToSend

State of the RTS pin

Description

You can configure `RequestToSend` to be on or off. If `RequestToSend` is on, the Request to Send (RTS) pin is asserted. If `RequestToSend` is off, the RTS pin is unasserted.

In normal usage, the RTS and Clear to Send (CTS) pins work together, and are used as standard handshaking pins for data transfer. In this case, RTS and CTS are automatically managed by the DTE and DCE. However, there is nothing in the RS-232 standard that requires the RTS pin must be used in any specific way. Therefore, if you manually configure the `RequestToSend` value, it is probably for nonstandard operations.

If your device does not use hardware handshaking in the standard way, and you need to manually configure `RequestToSend`, configure the `FlowControl` property to `none`. Otherwise, the `RequestToSend` value that you specify might not be honored. Refer to your device documentation to determine its specific pin behavior.

You can return the value of the CTS pin with the `PinStatus` property. Handshaking is described in “Controlling the Flow of Data: Handshaking” on page 23-102.

Characteristics

Read only	Never
Data type	String

Values

{on}	The RTS pin is asserted.
off	The RTS pin is unasserted.

Status

Indicate if the serial port object is connected to the device

Description

Status can be open or closed. If Status is closed, the serial port object is not connected to the device. If Status is open, the serial port object is connected to the device.

Before you can write or read data, you must connect the serial port object to the device with the `fopen` function. Use the `fclose` function to disconnect a serial port object from the device.

Characteristics

Read only	Always
Data type	String

Values

<code>{closed}</code>	The serial port object is not connected to the device.
<code>open</code>	The serial port object is connected to the device.

StopBits

Number of bits used to indicate the end of a byte

Description

You can configure `StopBits` to be 1, 1.5, or 2. If `StopBits` is 1, one stop bit is used to indicate the end of data transmission. If `StopBits` is 2, two stop bits are used to indicate the end of data transmission. If `StopBits` is 1.5, the stop bit is transferred for 150% of the normal time used to transfer one bit.

Note Both the computer and the peripheral device must be configured to transmit the same number of stop bits.

In addition to the stop bits, the serial data format on page 23-59 consists of a start bit, between five and eight data bits, and possibly a parity bit. You specify the number of data bits with the `DataBits` property, and the type of parity checking with the `Parity` property.

Characteristics

Read only	Never
Data type	Double

Values

{1}	One stop bit is transmitted to indicate the end of a byte.
1.5	The stop bit is transferred for 150% of the normal time used to transfer one bit.
2	Two stop bits are transmitted to indicate the end of a byte.

Tag

Label to associate with a serial port object

Description

You configure `Tag` to be a string value that uniquely identifies a serial port object.

`Tag` is particularly useful when constructing programs that would otherwise need to define the serial port object as a global variable, or pass the object as an argument between callback routines.

You can return the serial port object with the `instrfind` function by specifying the `Tag` property value.

Characteristics

Read only	Never
Data type	String

Values

The default value is an empty string.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');  
fopen(s)
```

You can assign `s` a unique label using `Tag`.

```
set(s, 'Tag', 'MySerialObj')
```

You can access `s` in the MATLAB workspace or in a file using the `instrfind` function and the `Tag` property value.

```
s1 = instrfind('Tag','MySerialObj');
```


Terminator

Terminator character

Description

You can configure `Terminator` to an integer value ranging from 0 to 127, which represents the ASCII code for the character, or you can configure `Terminator` to the ASCII character. For example, to configure `Terminator` to a carriage return, specify the value to be `CR` or 13. To configure `Terminator` to a linefeed, specify the value to be `LF` or 10. You can also set `Terminator` to `CR/LF` or `LF/CR`. If `Terminator` is `CR/LF`, the terminator is a carriage return followed by a line feed. If `Terminator` is `LF/CR`, the terminator is a linefeed followed by a carriage return. Note that there are no integer equivalents for these two values. Additionally, you can set `Terminator` to a 1-by-2 cell array. The first element of the cell is the read terminator and the second element of the cell array is the write terminator.

When performing a write operation using the `fprintf` function, all occurrences of `\n` are replaced with the `Terminator` property value. Note that `%s\n` is the default format for `fprintf`. A read operation with `fgetl`, `fgets`, or `fscanf` completes when the `Terminator` value is read. The terminator is ignored for binary operations.

You can also use the terminator to generate a bytes-available event when the `BytesAvailableFcnMode` is set to `terminator`.

Characteristics

Read only	Never
Data type	String

Values

An integer value ranging from 0 to 127, or the equivalent ASCII character. CR/LF and LF/CR are also accepted values. You specify different read and write terminators as a 1-by-2 cell array.

Timeout

Waiting time to complete a read or write operation

Description

You configure `Timeout` to be the maximum time (in seconds) to wait to complete a read or write operation.

If a time-out occurs, the read or write operation aborts. Additionally, if a time-out occurs during an asynchronous read or write operation, then:

- An error event is generated.
- The callback function specified for `ErrorFcn` is executed.

Characteristics

Read only	Never
Data type	Double

Values

The default value is 10 seconds.

TimerFcn

Specify the callback function to execute when a predefined period of time passes.

Description

You configure `TimerFcn` to execute a callback function when a timer event occurs. A timer event occurs when the time specified by the `TimerPeriod` property passes. Time is measured relative to when the serial port object is connected to the device with `fopen`.

Note A timer event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a timer event occurs, the record file records this information:

- The event type as `Timer`
- The time the event occurred using the format `day-month-year hour:minute:second:millisecond`

Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 23-96.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

TimerPeriod

Period of time between timer events

Description

TimerPeriod specifies the time, in seconds, that must pass before the callback function specified for TimerFcn is called. Time is measured relative to when the serial port object is connected to the device with fopen.

Some timer events might not be processed if your system is significantly slowed or if the TimerPeriod value is too small.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is 1 second. The minimum value is 0.01 second.

TransferStatus

Indicate if an asynchronous read or write operation is in progress

Description

TransferStatus can be `idle`, `read`, `write`, or `read&write`. If TransferStatus is `idle`, no asynchronous read or write operations are in progress. If TransferStatus is `read`, an asynchronous read operation is in progress. If TransferStatus is `write`, an asynchronous write operation is in progress. If TransferStatus is `read&write`, both an asynchronous read and an asynchronous write operation are in progress.

You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. While `readasync` is executing, TransferStatus might indicate that data is being read even though data is not filling the input buffer. If `ReadAsyncMode` is `continuous`, TransferStatus indicates that data is being read only when data is actually filling the input buffer.

You can execute an asynchronous read and an asynchronous write operation simultaneously because serial ports have separate read and write pins. For more information about synchronous and asynchronous read and write operations, see “Write and Read Data” on page 23-73.

Characteristics

Read only	Always
Data type	String

Values

<code>{idle}</code>	No asynchronous operations are in progress.
---------------------	---

read	An asynchronous read operation is in progress.
write	An asynchronous write operation is in progress.
read&write	Asynchronous read and write operations are in progress.

Type

Object type

Description

Type indicates the type of the object. Type is automatically defined after the serial port object is created with the `serial` function. The Type value is always `serial`.

Characteristics

Read only	Always
Data type	String

Values

Type is always `serial`. This value is automatically defined when the serial port object is created.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
```

The value of the Type property is `serial`, which is the object class.

```
s.Type  
ans =  
serial
```

You can also display the object class with the `whos` command.

```
Name      Size      Bytes  Class
```

```
s          1x1          644  serial object
Grand total is 18 elements using 644 bytes
```

UserData

Data you want to associate with a serial port object

Description

You configure `UserData` to store data that you want to associate with a serial port object. The object does not use this data directly, but you can access it using the `get` function or the dot notation.

Characteristics

Read only	Never
Data type	Any type

Values

The default value is an empty vector.

Example

Suppose you create the serial port object associated with the serial port COM1.

```
s = serial('COM1');
```

You can associate data with `s` by storing it in `UserData`.

```
coeff.a = 1.0;  
coeff.b = -1.25;  
s.UserData = coeff;
```

ValuesReceived

Total number of values read from the device

Description

ValuesReceived indicates the total number of values read from the device. The value is updated after each successful read operation, and is set to 0 after the `fopen` function is issued. If the terminator is read from the device, then this value is reflected by ValuesReceived.

If you are reading data asynchronously, use the `BytesAvailable` property to return the number of bytes currently available in the input buffer.

When performing a read operation, the received data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes. For more information about bytes and values, see “Bytes Versus Values” on page 23-59.

Characteristics

Read only	Always
Data type	Double

Values

The default value is 0.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');  
fopen(s)
```

If you write the RS232? command, and read back the response using fscanf, ValuesReceived is 17 because the instrument is configured to send the LF terminator.

```
fprintf(s, 'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
ans =
    17
```

ValuesSent

Total number of values written to the device

Description

ValuesSent indicates the total number of values written to the device. The value is updated after each successful write operation, and is set to 0 after the `fopen` function is issued. If you are writing the terminator, ValuesSent reflects this value.

If you are writing data asynchronously, use the BytesToOutput property to return the number of bytes currently in the output buffer.

When performing a write operation, the transmitted data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes. For more information about bytes and values, see “Bytes Versus Values” on page 23-59.

Characteristics

Read only	Always
Data type	Double

Values

The default value is 0.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');  
fopen(s)
```

If you write the `*IDN?` command using the `fprintf` function, `ValuesSent` is 6 because the default data format is `%s\n`, and the terminator was written.

```
fprintf(s, '*IDN?')  
s.ValuesSent  
ans =  
    6
```


Hardware Support

- “Support Package Installation” on page 24-2
- “Change Your Support Package Directory” on page 24-4

Support Package Installation

In this section...

“What Is a Support Package?” on page 24-2

“Install Support Packages” on page 24-2

What Is a Support Package?

A support package is an add-on that enables you to use a MathWorks product with specific third-party hardware and software.

For example, support packages available for use with MATLAB include:

- Arduino® Hardware
- LEGO® MINDSTORMS® EV3 Hardware
- Raspberry Pi™ Hardware
- USB Webcam

For a complete list of available support packages, see Hardware Support.

Install Support Packages

You install support packages using the MATLAB **Add-Ons** menu. You can also use **Add-Ons** to update installed support package software or update the firmware on third-party hardware.

To install support packages, on the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

See Also

More About

- “Get and Manage Add-Ons”

External Websites

- Hardware Support Catalog

Change Your Support Package Directory

When installing support packages, the root directory is set to a default location for your specific system.

- For Windows, the default root directory is :

```
'C:\ProgramData\MATLAB\SupportPackage\<<release>'
```

where `<release>` is your specific MATLAB release.

If the default directory already exists and is not empty, the root directory becomes `'C:\ProgramData\MATLAB\SupportPackages\<<release>_1'`.

- For Unix, the default root directory is:

```
'/home/user/Documents/MATLAB/SupportPackages/<<release>'
```

where `<release>` is your specific MATLAB release.

If the computed default directory already exists and is not empty, it becomes `'/home/user/Documents/MATLAB/SupportPackages\<<release>_1'`.

You can change the location if your application requires a custom directory for installation. You must have administration privileges and be able to write to this custom folder. In order to set the support package root and verify the change, follow this example:

Note You must have administration privileges for the custom folder you specify.

```
matlabshared.supportpkg.setSupportPackageRoot('C:\MATLAB\CustomSupportPackageRoot')
```

```
matlabshared.supportpkg.getSupportPackageRoot
```

```
ans =
```

```
C:\MATLAB\CustomSupportPackageRoot
```

When setting the support package root, `savepath` is called to add the location to the MATLAB search path. To ensure the path is loaded correctly, start MATLAB, run the `setSupportPackageRoot` function, and then restart MATLAB.

Note It is not recommended you use multiple different directories to install separate support packages. Only the support packages installed on the current directory can be used.
