<table>
<thead>
<tr>
<th>Date</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 1996</td>
<td>First printing</td>
<td>New for MATLAB 5 (release 8)</td>
</tr>
<tr>
<td>July 1997</td>
<td>Online only</td>
<td>Revised for MATLAB 5.1 (Release 9)</td>
</tr>
<tr>
<td>January 1998</td>
<td>Second printing</td>
<td>Revised for MATLAB 5.2 (Release 10)</td>
</tr>
<tr>
<td>October 1998</td>
<td>Third printing</td>
<td>Revised for MATLAB 5.3 (Release 11)</td>
</tr>
<tr>
<td>November 2000</td>
<td>Fourth printing</td>
<td>Revised and renamed for MATLAB 6.0 (Release 12)</td>
</tr>
<tr>
<td>June 2001</td>
<td>Online only</td>
<td>Revised for MATLAB 6.1 (Release 12.1)</td>
</tr>
<tr>
<td>July 2002</td>
<td>Online only</td>
<td>Revised for MATLAB 6.5 (Release 13)</td>
</tr>
<tr>
<td>January 2003</td>
<td>Online only</td>
<td>Revised for MATLAB 6.5.1 (Release 13SP1)</td>
</tr>
<tr>
<td>June 2004</td>
<td>Online only</td>
<td>Revised for MATLAB 7.0 (Release 14)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Online only</td>
<td>Revised for MATLAB 7.0.1 (Release 14SP1)</td>
</tr>
<tr>
<td>September 2005</td>
<td>Online only</td>
<td>Revised for MATLAB 7.1 (Release 14SP3)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Revised for MATLAB 7.2 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for MATLAB 7.3 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for MATLAB 7.4 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for MATLAB 7.5 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for MATLAB 7.6 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for MATLAB 7.7 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for MATLAB 7.8 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for MATLAB 7.9 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for MATLAB 7.10 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for MATLAB 7.11 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for MATLAB 7.12 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for MATLAB 7.13 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for MATLAB 7.14 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for MATLAB 8.0 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for MATLAB 8.1 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for MATLAB 8.2 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for MATLAB 8.3 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for MATLAB 8.4 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for MATLAB 8.5 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for MATLAB 8.6 (Release 2015b)</td>
</tr>
</tbody>
</table>
Install Supported Python Implementation .................................. 1-3
  Install Python Version 2.7 ............................................. 1-3
  Install Python Version 3.3 or 3.4 ..................................... 1-3

Call Python from MATLAB .................................................. 1-4

Call User-Defined Python Module ....................................... 1-10

Use Python Numeric Types in MATLAB ................................. 1-12

Call Python Methods with Numeric Arguments ....................... 1-13

Default Numeric Types ..................................................... 1-14

Use Python array Types in MATLAB .................................... 1-15

Pass MATLAB String to Python Method ................................. 1-16

Use Python str Type in MATLAB ......................................... 1-17

Pass MATLAB Backslash Control Character ............................ 1-18

Create Python list Variable .............................................. 1-19

Use Python list Type in MATLAB ........................................ 1-20

Use Python List of Numeric Types in MATLAB ....................... 1-22

Pass Cell Array as Python Sequence Type ............................. 1-23

Read Element of Nested list Type ...................................... 1-25
Use Python tuple Type in MATLAB .......................... 1-26
Create Python tuple Variable .............................. 1-27
Create Singleton Python tuple Variable .................... 1-28
Create Python dict Variable .............................. 1-29
Pass dict Argument to Python Method ....................... 1-30
Use Python dict Type in MATLAB ......................... 1-31
Convert Python dict Type to MATLAB Structure .......... 1-33
Pass Keyword Arguments .................................. 1-34
Pass Python Function to Python map Function .......... 1-35
Index into Python String ................................ 1-36
Index into Python List .................................. 1-37
Index into Python Tuple ................................ 1-39
Index into Python dict .................................. 1-40
Use Python List as Values in for Loop .................... 1-41
Display Stepped Range of Elements ......................... 1-42
Access Elements in Python Container Types ............... 1-43
  Sequence Types ........................................ 1-43
  Mapping Types ....................................... 1-44
  Size and Dimensions .................................. 1-44
  Array Support ....................................... 1-45
  Use Zero-Based Indexing for Python Functions .......... 1-45
View Python Numeric Values ................................ 1-46
  Why Do I See Properties When I Display a Number? ... 1-46
  What Is the L Character Attached to a Number? .......... 1-46
Call Methods on Python Variables .......................... 1-48
Help for Python Functions ........................................ 1-73
Handle Python Exceptions ........................................ 1-74
Troubleshooting Error Messages ............................... 1-75
  Python Error: Python class: message ....................... 1-75
  Python Module Errors ........................................... 1-75
  Errors Converting Python Data ............................... 1-76
Using Python Data in MATLAB ................................. 1-77
Call Python eval Function ........................................ 1-78
Precedence Order of Methods and Functions ............... 1-80
Python Function Arguments ..................................... 1-81
  Positional Arguments ......................................... 1-81
  Keyword Arguments .......................................... 1-81
  Optional Arguments .......................................... 1-82

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

Custom Applications to Access MAT-Files ..................... 2-2
  Why Write Custom Applications? ............................ 2-2
  MAT-File Interface Library ................................ 2-3
  Exchanging Data Files Between Platforms ............... 2-4
MAT-File Library and Include Files .......................... 2-5
  MAT-Function Include Files ................................ 2-5
  MAT-Function Libraries .................................... 2-5
  Example Files .................................................. 2-6
What You Need to Build Custom Applications ................ 2-7
Copy External Data into MAT-File Format with Standalone Programs ............................................. 2-8
  Overview of matimport.c Example ......................... 2-8
  Declare Variables for External Data ...................... 2-9
<table>
<thead>
<tr>
<th>Content</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create mxArray Variables</td>
<td>2-9</td>
</tr>
<tr>
<td>Create MATLAB Variable Names</td>
<td>2-10</td>
</tr>
<tr>
<td>Read External Data into mxArray Data</td>
<td>2-10</td>
</tr>
<tr>
<td>Create and Open MAT-File</td>
<td>2-11</td>
</tr>
<tr>
<td>Write mxArray Data to File</td>
<td>2-11</td>
</tr>
<tr>
<td>Clean Up</td>
<td>2-11</td>
</tr>
<tr>
<td>Build the Application</td>
<td>2-11</td>
</tr>
<tr>
<td>Create the MAT-File</td>
<td>2-11</td>
</tr>
<tr>
<td>Import Data into MATLAB</td>
<td>2-12</td>
</tr>
<tr>
<td><strong>Create MAT-File in C or C++</strong></td>
<td>2-13</td>
</tr>
<tr>
<td>Create MAT-File in C</td>
<td>2-13</td>
</tr>
<tr>
<td>Create MAT-File in C++</td>
<td>2-13</td>
</tr>
<tr>
<td><strong>Read MAT-File in C/C++</strong></td>
<td>2-14</td>
</tr>
<tr>
<td><strong>Create MAT-File in Fortran</strong></td>
<td>2-15</td>
</tr>
<tr>
<td><strong>Read MAT-File in Fortran</strong></td>
<td>2-16</td>
</tr>
<tr>
<td><strong>Work with mxArrays</strong></td>
<td>2-17</td>
</tr>
<tr>
<td>Read Structures from a MAT-File</td>
<td>2-17</td>
</tr>
<tr>
<td>Read Cell Arrays from a MAT-File</td>
<td>2-18</td>
</tr>
<tr>
<td><strong>Table of MAT-File Source Code Files</strong></td>
<td>2-19</td>
</tr>
<tr>
<td><strong>Build on Mac and Linux Operating Systems</strong></td>
<td>2-21</td>
</tr>
<tr>
<td>Setting Run-Time Library Path</td>
<td>2-21</td>
</tr>
<tr>
<td>Building the Application</td>
<td>2-22</td>
</tr>
<tr>
<td><strong>Build on Windows Operating Systems</strong></td>
<td>2-23</td>
</tr>
<tr>
<td><strong>Share MAT-File Applications</strong></td>
<td>2-24</td>
</tr>
</tbody>
</table>

### Calling C Shared Library Functions from MATLAB

<table>
<thead>
<tr>
<th>Content</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call Functions in Shared Libraries</strong></td>
<td>3-2</td>
</tr>
<tr>
<td>What Is a Shared Library?</td>
<td>3-2</td>
</tr>
<tr>
<td>Load and Unload Library</td>
<td>3-3</td>
</tr>
</tbody>
</table>
View Library Functions .................................................. 3-4
Invoke Library Functions ............................................... 3-6

Limitations to Shared Library Support .............................. 3-8
  MATLAB Supports C Library Routines ............................... 3-8
  Workarounds for Loading C++ Libraries ........................... 3-8
  Limitations Using printf Function ................................. 3-9
  Bit Fields .................................................................. 3-9
  Enum Declarations ....................................................... 3-10
  Unions Not Supported .................................................. 3-10
  Compiler Dependencies .................................................. 3-11
  Limitations Using Pointers ............................................ 3-11
  Functions with Variable Number of Input Arguments Not
  Supported ................................................................. 3-12

Limitations Using Structures ......................................... 3-13
  MATLAB Returns Pointers to Structures ........................... 3-13
  Structure Cannot Contain Pointers to Other Structures ...... 3-13
  Requirements for MATLAB Structure Arguments ............... 3-13
  Requirements for C struct Field Names ......................... 3-13

Module Not Found Error .................................................. 3-15

No Matching Signature Error ............................................ 3-16

MATLAB Terminates Unexpectedly When Calling Function in
Shared Library ............................................................. 3-17

Pass Arguments to Shared Library Functions ..................... 3-18
  C and MATLAB Equivalent Types .................................... 3-18
  How MATLAB Displays Function Signatures ..................... 3-20
  NULL Pointer .............................................................. 3-21
  Manually Convert Data Passed to Functions ..................... 3-21

Shared Library shrlibsample ............................................ 3-23

Pass String Arguments .................................................. 3-24
  stringToUpper Function .............................................. 3-24
  Convert MATLAB Character Array to Uppercase ................. 3-24

Pass Structures ................................................................ 3-26
  addStructFields and addStructByRef Functions ................. 3-26
  Add Values of Fields in Structure .................................... 3-27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconvert MATLAB Structure Before Adding Values</td>
<td>3-28</td>
</tr>
<tr>
<td>Autoconvert Structure Arguments</td>
<td>3-29</td>
</tr>
<tr>
<td>Pass Pointer to Structure</td>
<td>3-30</td>
</tr>
<tr>
<td><strong>Pass Enumerated Types</strong></td>
<td>3-32</td>
</tr>
<tr>
<td>readEnum Function</td>
<td>3-32</td>
</tr>
<tr>
<td>Display Enumeration Values</td>
<td>3-32</td>
</tr>
<tr>
<td><strong>Pass Pointers</strong></td>
<td>3-34</td>
</tr>
<tr>
<td>multDoubleRef Function</td>
<td>3-34</td>
</tr>
<tr>
<td>Pass Pointer of Type double</td>
<td>3-34</td>
</tr>
<tr>
<td>Create Pointer Offset from Existing lib.pointer Object</td>
<td>3-35</td>
</tr>
<tr>
<td>Multilevel Pointers</td>
<td>3-36</td>
</tr>
<tr>
<td>allocateStruct and deallocateStruct Functions</td>
<td>3-36</td>
</tr>
<tr>
<td>Pass Multilevel Pointer</td>
<td>3-37</td>
</tr>
<tr>
<td>Return Array of Strings</td>
<td>3-37</td>
</tr>
<tr>
<td><strong>Pass Arrays</strong></td>
<td>3-39</td>
</tr>
<tr>
<td>print2darray Function</td>
<td>3-39</td>
</tr>
<tr>
<td>Convert MATLAB Array to C-Style Dimensions</td>
<td>3-39</td>
</tr>
<tr>
<td>multDoubleArray Function</td>
<td>3-40</td>
</tr>
<tr>
<td>Preserve 3-D MATLAB Array</td>
<td>3-41</td>
</tr>
<tr>
<td><strong>Iterate Through an Array</strong></td>
<td>3-43</td>
</tr>
<tr>
<td>Create Cell Array from lib.pointer Object</td>
<td>3-43</td>
</tr>
<tr>
<td>Perform Pointer Arithmetic on Structure Array</td>
<td>3-44</td>
</tr>
<tr>
<td><strong>Pointer Arguments</strong></td>
<td>3-46</td>
</tr>
<tr>
<td>Pointer Arguments in C Functions</td>
<td>3-46</td>
</tr>
<tr>
<td>Put String into Void Pointer</td>
<td>3-46</td>
</tr>
<tr>
<td>Memory Allocation for External Library</td>
<td>3-47</td>
</tr>
<tr>
<td><strong>Structure Arguments</strong></td>
<td>3-49</td>
</tr>
<tr>
<td>Structure Argument Requirements</td>
<td>3-49</td>
</tr>
<tr>
<td>Find Structure Field Names</td>
<td>3-49</td>
</tr>
<tr>
<td>Strategies for Passing Structures</td>
<td>3-49</td>
</tr>
<tr>
<td><strong>Explore libstruct Objects</strong></td>
<td>3-51</td>
</tr>
<tr>
<td><strong>MATLAB Prototype Files</strong></td>
<td>3-52</td>
</tr>
<tr>
<td>When to Use Prototype Files</td>
<td>3-52</td>
</tr>
<tr>
<td>How to Create Prototype Files</td>
<td>3-52</td>
</tr>
<tr>
<td>How to Specify Thunk Files</td>
<td>3-53</td>
</tr>
</tbody>
</table>
Deploy Applications That Use `loadlibrary` .......................... 3-53
`loadlibrary` in Parallel Computing Environment .................. 3-53
Change Function Signature ........................................ 3-53
Rename Library Function ........................................ 3-53
Load Subset of Functions in Library .............................. 3-53
Call Function with Variable Number of Arguments .............. 3-54

Intro to MEX-Files

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introducing MEX Files</td>
<td>4-3</td>
</tr>
<tr>
<td>Using MEX Files</td>
<td>4-4</td>
</tr>
<tr>
<td>MEX File Placement</td>
<td>4-5</td>
</tr>
<tr>
<td>MEX Files on Windows Network Drives</td>
<td>4-5</td>
</tr>
<tr>
<td>Use Help Files with MEX Files</td>
<td>4-6</td>
</tr>
<tr>
<td>MATLAB Data</td>
<td>4-7</td>
</tr>
<tr>
<td>The MATLAB Array</td>
<td>4-7</td>
</tr>
<tr>
<td>Lifecycle of mxArray</td>
<td>4-7</td>
</tr>
<tr>
<td>Data Storage</td>
<td>4-8</td>
</tr>
<tr>
<td>MATLAB Types</td>
<td>4-10</td>
</tr>
<tr>
<td>Sparse Matrices</td>
<td>4-11</td>
</tr>
<tr>
<td>Using Data Types</td>
<td>4-12</td>
</tr>
<tr>
<td>Testing for Most-Derived Class</td>
<td>4-15</td>
</tr>
<tr>
<td>Testing for a Category of Types</td>
<td>4-15</td>
</tr>
<tr>
<td>Another Test for Built-In Types</td>
<td>4-16</td>
</tr>
<tr>
<td>Build MEX File</td>
<td>4-17</td>
</tr>
<tr>
<td>Linking Multiple Files</td>
<td>4-18</td>
</tr>
<tr>
<td>What You Need to Build MEX Files</td>
<td>4-19</td>
</tr>
<tr>
<td>Change Default Compiler</td>
<td>4-20</td>
</tr>
<tr>
<td>Windows Systems</td>
<td>4-20</td>
</tr>
<tr>
<td>Mac and Linux Systems</td>
<td>4-21</td>
</tr>
</tbody>
</table>
## Document Build Information in the MEX File

- Page 4-56

## Version Compatibility

- Page 4-58

## Getting Help When MEX Fails

- Errors Finding Supported Compiler: Page 4-59
- Errors Building MEX Function: Page 4-59
- Preview mex Build Commands: Page 4-60

## Understanding MEX File Problems

- Problem 1 — Compiling a Source MEX File Fails: Page 4-63
- Problem 2 — Compiling Your Own Program Fails: Page 4-63
- Problem 3 — Binary MEX File Load Errors: Page 4-64
- Problem 4 — Segmentation Fault: Page 4-65
- Problem 5 — Program Generates Incorrect Results: Page 4-65

## Compiler- and Platform-Specific Issues

- Linux gcc Compiler Version Error: Page 4-66
- Linux gcc -fPIC Errors: Page 4-66

## Memory Management Issues

- Overview: Page 4-67
- Improperly Destroying an mxArray: Page 4-68
- Incorrectly Constructing a Cell or Structure mxArray: Page 4-68
- Creating a Temporary mxArray with Improper Data: Page 4-69
- Creating Potential Memory Leaks: Page 4-70
- Improperly Destroying a Structure: Page 4-70
- Destroying Memory in a C++ Class Destructor: Page 4-71

## Compiler Errors in Fortran MEX Files

- Page 4-73

## C/C++ MEX-Files

### Components of MEX File

- Page 5-3
  - mexFunction Gateway Routine: Page 5-3
  - Naming the MEX File: Page 5-3
  - Required Parameters: Page 5-3
  - Managing Input and Output Parameters: Page 5-4
  - Validating Inputs: Page 5-4
Computational Routine ........................................ 5-5

MATLAB API Libraries ........................................ 5-6
  Matrix Library ............................................. 5-6
  MEX Library ............................................... 5-6
  Preprocessor Macros ....................................... 5-6

User Messages .................................................. 5-8

Error Handling .................................................. 5-9

Data Flow in MEX Files ........................................ 5-10
  Showing Data Input and Output .............................. 5-10
  Gateway Routine Data Flow Diagram ......................... 5-11

Creating C++ MEX Files ........................................ 5-13
  Creating Your C++ Source File .............................. 5-13
  Compiling and Linking ..................................... 5-13
  Memory Considerations for Class Destructors ............. 5-13
  Use mexPrintf to Print to MATLAB Command Window ...... 5-14

C++ Class in MEX Files ......................................... 5-15

Handle Files with C++ .......................................... 5-16
  C++ Example ............................................... 5-16
  C Example .................................................. 5-16

Create C Source MEX File ...................................... 5-18

Table of MEX File Source Code Files ......................... 5-24

Choose a C++ Compiler ......................................... 5-28
  Select Microsoft Visual Studio Compiler .................. 5-28
  Select MinGW-w64 Compiler ................................ 5-28

Set Up C/C++ Examples ......................................... 5-30

Pass Scalar Values ............................................. 5-31
  Pass Scalar as Matrix .................................... 5-31
  Pass Scalar by Value .................................... 5-32

Pass Strings .................................................... 5-34
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handling Strings in C/C++</td>
<td>5-36</td>
</tr>
<tr>
<td>How MATLAB Represents Strings in MEX-Files</td>
<td>5-36</td>
</tr>
<tr>
<td>Character Encoding and Multibyte Encoding Schemes</td>
<td>5-36</td>
</tr>
<tr>
<td>Converting MATLAB String to C-Style String</td>
<td>5-37</td>
</tr>
<tr>
<td>Converting C-Style String to MATLAB String</td>
<td>5-37</td>
</tr>
<tr>
<td>Returning Modified Input String</td>
<td>5-37</td>
</tr>
<tr>
<td>Memory Management</td>
<td>5-37</td>
</tr>
<tr>
<td>Pass Multiple Inputs or Outputs</td>
<td>5-39</td>
</tr>
<tr>
<td>Pass Structures and Cell Arrays</td>
<td>5-41</td>
</tr>
<tr>
<td>Create 2-D Cell Array</td>
<td>5-42</td>
</tr>
<tr>
<td>Fill mxArray</td>
<td>5-43</td>
</tr>
<tr>
<td>Options</td>
<td>5-43</td>
</tr>
<tr>
<td>Copying Data Directly into an mxArray</td>
<td>5-43</td>
</tr>
<tr>
<td>Pointing to Data</td>
<td>5-43</td>
</tr>
<tr>
<td>Prompt User for Input</td>
<td>5-45</td>
</tr>
<tr>
<td>Handle Complex Data</td>
<td>5-46</td>
</tr>
<tr>
<td>Handle 8-, 16-, and 32-Bit Data</td>
<td>5-47</td>
</tr>
<tr>
<td>Manipulate Multidimensional Numerical Arrays</td>
<td>5-48</td>
</tr>
<tr>
<td>Handle Sparse Arrays</td>
<td>5-50</td>
</tr>
<tr>
<td>Call MATLAB Functions from C/C++ MEX Files</td>
<td>5-51</td>
</tr>
<tr>
<td>Debugging on Microsoft Windows Platforms</td>
<td>5-52</td>
</tr>
<tr>
<td>Notes on Debugging</td>
<td>5-53</td>
</tr>
<tr>
<td>Debugging on Linux Platforms</td>
<td>5-54</td>
</tr>
<tr>
<td>Debugging on Mac Platforms</td>
<td>5-56</td>
</tr>
<tr>
<td>Using Xcode</td>
<td>5-56</td>
</tr>
<tr>
<td>Using LLDB</td>
<td>5-58</td>
</tr>
<tr>
<td>Handling Large mxArray</td>
<td>5-61</td>
</tr>
<tr>
<td>Using the 64-Bit API</td>
<td>5-61</td>
</tr>
<tr>
<td>Building the Binary MEX File</td>
<td>5-63</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Example</td>
<td>5-63</td>
</tr>
<tr>
<td>Caution Using Negative Values</td>
<td>5-63</td>
</tr>
<tr>
<td>Building Cross-Platform Applications</td>
<td>5-64</td>
</tr>
<tr>
<td><strong>Memory Management</strong></td>
<td>5-65</td>
</tr>
<tr>
<td>Automatic Cleanup of Temporary Arrays</td>
<td>5-65</td>
</tr>
<tr>
<td>Example</td>
<td>5-66</td>
</tr>
<tr>
<td>Persistent Arrays</td>
<td>5-66</td>
</tr>
<tr>
<td><strong>Handling Large File I/O</strong></td>
<td>5-68</td>
</tr>
<tr>
<td>Prerequisites to Using 64-Bit I/O</td>
<td>5-68</td>
</tr>
<tr>
<td>Specifying Constant Literal Values</td>
<td>5-70</td>
</tr>
<tr>
<td>Opening a File</td>
<td>5-70</td>
</tr>
<tr>
<td>Printing Formatted Messages</td>
<td>5-71</td>
</tr>
<tr>
<td>Replacing fseek and ftell with 64-Bit Functions</td>
<td>5-71</td>
</tr>
<tr>
<td>Determining the Size of an Open File</td>
<td>5-72</td>
</tr>
<tr>
<td>Determining the Size of a Closed File</td>
<td>5-73</td>
</tr>
<tr>
<td><strong>Install MinGW-w64 Compiler</strong></td>
<td>5-74</td>
</tr>
<tr>
<td>Installing Compiler from Add-Ons Menu</td>
<td>5-74</td>
</tr>
<tr>
<td>Building yprime.c Example</td>
<td>5-75</td>
</tr>
<tr>
<td>MinGW Installation Folder Cannot Contain Space</td>
<td>5-75</td>
</tr>
<tr>
<td>Updating MEX Files to Use MinGW Compiler</td>
<td>5-75</td>
</tr>
<tr>
<td><strong>Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64</strong></td>
<td>5-77</td>
</tr>
<tr>
<td>Do Not Link to Library Files Compiled with Non-MinGW Compilers</td>
<td>5-77</td>
</tr>
<tr>
<td>MinGW Compiler Not Setup for Use with MEX</td>
<td>5-77</td>
</tr>
<tr>
<td>MinGW Installation Folder Cannot Contain Space</td>
<td>5-78</td>
</tr>
<tr>
<td>MEX Command Does not Choose MinGW</td>
<td>5-78</td>
</tr>
<tr>
<td>Manually Configure MinGW for MATLAB</td>
<td>5-78</td>
</tr>
<tr>
<td>Potential Memory Leak Inside C++ MEX Files on Using MEX Exceptions</td>
<td>5-79</td>
</tr>
<tr>
<td>Unhandled Explicit Exceptions in C++ MEX Files Unexpectedly Terminate MATLAB</td>
<td>5-80</td>
</tr>
<tr>
<td>Out of Memory Error for Variables Containing Large Amounts of Data</td>
<td>5-81</td>
</tr>
</tbody>
</table>
Components of Fortran MEX File ........................................... 6-2
   mexFunction Gateway Routine .................................. 6-2
   Naming the MEX File ............................................. 6-2
   Difference Between .f and .F Files ......................... 6-2
   Required Parameters ........................................... 6-3
   Managing Input and Output Parameters .................... 6-3
   Validating Inputs ............................................. 6-4
   Computational Routine ....................................... 6-4

MATLAB Fortran API Libraries ........................................ 6-5
   Matrix Library .................................................. 6-5
   MEX Library ................................................... 6-5
   Preprocessor Macros ......................................... 6-5
   Using the Fortran %val Construct ......................... 6-6

Data Flow in Fortran MEX Files ..................................... 6-8
   Showing Data Input and Output ............................ 6-8
   Gateway Routine Data Flow Diagram .................... 6-9

User Messages ............................................................ 6-11

Error Handling ............................................................ 6-12

Build Fortran MEX File .............................................. 6-13

Create Fortran Source MEX File ................................... 6-14

Set Up Fortran Examples ............................................ 6-19

Pass Scalar Values .................................................... 6-20

Pass Strings ............................................................. 6-21

Pass Arrays of Strings .............................................. 6-22

Pass Matrices ............................................................ 6-23

Pass Integers ............................................................. 6-24
Pass Multiple Inputs or Outputs .......................... 6-25
Handle Complex Data ....................................... 6-26
Dynamically Allocate Memory .............................. 6-27
Handle Sparse Matrices ..................................... 6-28
Call MATLAB Functions from Fortran MEX Files ....... 6-29
Debug Fortran Source MEX-Files ......................... 6-31
  Notes on Debugging ..................................... 6-31
  Debugging on Microsoft Windows Platforms .......... 6-31
  Debugging on Linux Platforms ......................... 6-31
Handling Large mxArrays ................................. 6-34
  Using the 64-Bit API .................................... 6-34
  Building the Binary MEX File ......................... 6-36
  Caution Using Negative Values ....................... 6-36
  Building Cross-Platform Applications ............... 6-36
Memory Management ........................................ 6-37
MATLAB Supports Fortran 77 .............................. 6-38

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

Introducing MATLAB Engine API for C/C++ and Fortran .. 7-2
  Communicating with MATLAB Software ................. 7-3
Callbacks in Applications ................................ 7-4
Call MATLAB Functions from C/C++ Applications ........ 7-5
Call MATLAB Functions from Fortran Applications ....... 7-7
Attach to Existing MATLAB Sessions .................... 7-9
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Windows Engine Application</td>
<td>7-11</td>
</tr>
<tr>
<td>Run Windows Engine Application</td>
<td>7-13</td>
</tr>
<tr>
<td>Set Run-Time Library Path on Windows Systems</td>
<td>7-14</td>
</tr>
<tr>
<td>- Change Path Each Time You Run the Application</td>
<td>7-14</td>
</tr>
<tr>
<td>- Permanently Change Path</td>
<td>7-14</td>
</tr>
<tr>
<td>- Troubleshooting</td>
<td>7-15</td>
</tr>
<tr>
<td>Register MATLAB as a COM Server</td>
<td>7-16</td>
</tr>
<tr>
<td>Build Linux Engine Application</td>
<td>7-17</td>
</tr>
<tr>
<td>Run Linux Engine Application</td>
<td>7-18</td>
</tr>
<tr>
<td>Set Run-Time Library Path on Mac and Linux Systems</td>
<td>7-19</td>
</tr>
<tr>
<td>- C Shell</td>
<td>7-19</td>
</tr>
<tr>
<td>- Bourne Shell</td>
<td>7-20</td>
</tr>
<tr>
<td>Build Engine Applications with IDE</td>
<td>7-21</td>
</tr>
<tr>
<td>- Configuring the IDE</td>
<td>7-21</td>
</tr>
<tr>
<td>- Engine Include Files</td>
<td>7-21</td>
</tr>
<tr>
<td>- Engine Libraries</td>
<td>7-21</td>
</tr>
<tr>
<td>- Library Files Required by libeng</td>
<td>7-22</td>
</tr>
<tr>
<td>Can't Start MATLAB Engine</td>
<td>7-24</td>
</tr>
<tr>
<td>Debug MATLAB Functions Used in Engine Applications</td>
<td>7-25</td>
</tr>
<tr>
<td>Multithreaded Applications</td>
<td>7-26</td>
</tr>
<tr>
<td>User Input Not Supported</td>
<td>7-27</td>
</tr>
<tr>
<td>Getting Started</td>
<td>7-28</td>
</tr>
</tbody>
</table>

# MATLAB Engine for Python Topics

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Started with MATLAB Engine for Python</td>
<td>8-2</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Install MATLAB Engine for Python</td>
<td>8-5</td>
</tr>
<tr>
<td>Verify Python and MATLAB Installations</td>
<td>8-5</td>
</tr>
<tr>
<td>Install Engine</td>
<td>8-5</td>
</tr>
<tr>
<td>Build Engine in Nondefault Folder</td>
<td>8-6</td>
</tr>
<tr>
<td>Install Engine in Nondefault Folder</td>
<td>8-6</td>
</tr>
<tr>
<td>Install Engine in Your Home Folder</td>
<td>8-6</td>
</tr>
<tr>
<td>Set Run-Time Paths To Python Code</td>
<td>8-7</td>
</tr>
<tr>
<td>Start and Stop MATLAB Engine for Python</td>
<td>8-8</td>
</tr>
<tr>
<td>Start MATLAB Engine for Python</td>
<td>8-8</td>
</tr>
<tr>
<td>Run Multiple Engines</td>
<td>8-8</td>
</tr>
<tr>
<td>Stop MATLAB Engine</td>
<td>8-8</td>
</tr>
<tr>
<td>Start Engine with Startup Options</td>
<td>8-9</td>
</tr>
<tr>
<td>Connect Python to Running MATLAB Session</td>
<td>8-10</td>
</tr>
<tr>
<td>Connect to Shared MATLAB Session</td>
<td>8-10</td>
</tr>
<tr>
<td>Connect to Multiple Shared MATLAB Sessions</td>
<td>8-11</td>
</tr>
<tr>
<td>Start Shared MATLAB Sessions with Startup Options</td>
<td>8-11</td>
</tr>
<tr>
<td>Call MATLAB Functions from Python</td>
<td>8-13</td>
</tr>
<tr>
<td>Return Output Argument from MATLAB Function</td>
<td>8-13</td>
</tr>
<tr>
<td>Return Multiple Output Arguments from MATLAB Function</td>
<td>8-13</td>
</tr>
<tr>
<td>Return No Output Arguments from MATLAB Function</td>
<td>8-13</td>
</tr>
<tr>
<td>Stop Execution of Function</td>
<td>8-14</td>
</tr>
<tr>
<td>Call MATLAB Functions Asynchronously from Python</td>
<td>8-15</td>
</tr>
<tr>
<td>Call User Script and Function from Python</td>
<td>8-16</td>
</tr>
<tr>
<td>Redirect Standard Output and Error to Python</td>
<td>8-18</td>
</tr>
<tr>
<td>Use MATLAB Handle Objects in Python</td>
<td>8-19</td>
</tr>
<tr>
<td>Use MATLAB Engine Workspace in Python</td>
<td>8-21</td>
</tr>
<tr>
<td>Pass Data to MATLAB from Python</td>
<td>8-22</td>
</tr>
<tr>
<td>Python Type to MATLAB Scalar Type Mapping</td>
<td>8-22</td>
</tr>
<tr>
<td>Python Container to MATLAB Array Type Mapping</td>
<td>8-22</td>
</tr>
<tr>
<td>Unsupported Python Types</td>
<td>8-23</td>
</tr>
<tr>
<td>Handle Data Returned from MATLAB to Python</td>
<td>8-24</td>
</tr>
<tr>
<td>MATLAB Scalar Type to Python Type Mapping</td>
<td>8-24</td>
</tr>
<tr>
<td>MATLAB Array Type to Python Type Mapping</td>
<td>8-25</td>
</tr>
</tbody>
</table>
Unsupported MATLAB Types ........................................... 8-25

MATLAB Arrays as Python Variables ............................... 8-27
Create MATLAB Arrays in Python ................................. 8-27
MATLAB Array Attributes and Methods in Python .......... 8-29
Multidimensional MATLAB Arrays in Python ................. 8-29
Index Into MATLAB Arrays in Python ......................... 8-29
Slice MATLAB Arrays in Python ................................. 8-30
Reshape MATLAB Arrays in Python .............................. 8-31

Use MATLAB Arrays in Python ...................................... 8-32

Sort and Plot MATLAB Data from Python ....................... 8-34

Get Help for MATLAB Functions from Python ................. 8-38
How to Find MATLAB Help ......................................... 8-38
Open MATLAB Help Browser from Python ..................... 8-38
Display MATLAB Help at Python Prompt ....................... 8-39

Default Numeric Types in MATLAB and Python ............... 8-40

System Requirements for MATLAB Engine for Python ....... 8-42
Python Version Support ............................................ 8-42
64-bit or 32-bit Versions of Python and MATLAB .......... 8-43
Requirements for Building Python from Source .............. 8-43

Limitations to MATLAB Engine for Python .................... 8-44

Troubleshoot MATLAB Errors in Python ....................... 8-45
MATLAB Errors in Python ........................................ 8-45
MatlabExecutionError: Undefined Function ................. 8-45
SyntaxError: Expression Not Valid Target .................... 8-46
SyntaxError: Invalid Syntax .................................... 8-46

Using Java Libraries from MATLAB

Call Method on Java Object ...................................... 9-2
# Java Libraries

Java Software Is Integral to MATLAB ........................................ 9-3
When to Use Java Libraries in MATLAB .................................. 9-3
To Learn More About Java Programming Language ................... 9-4
Platform Support for JVM Software ......................................... 9-4

## Bring Java Classes into MATLAB Workspace

Introduction .................................................................................. 9-5
Defining New Java Classes ....................................................... 9-5
Java Class Path ........................................................................... 9-5
Making Java Classes Available in MATLAB Workspace ............. 9-7
Loading Java Class Definitions .................................................. 9-9
Simplifying Java Class Names Using import Function ................. 9-9
Locating Native Method Libraries ............................................. 9-10

## Convert Java String to Uppercase

................................................................. 9-12

## Use Class in Java JAR File on Static Class Path

................................................................. 9-13

## Call User-Defined Java Class on Dynamic Class Path

................................................................. 9-14

## Java Objects

Overview ....................................................................................... 9-15
Constructing Java Objects ....................................................... 9-15
Concatenating Java Objects ...................................................... 9-17
Saving and Loading Java Objects to MAT-Files ......................... 9-18
Finding the Public Data Fields of an Object ................................. 9-19
Accessing Private and Public Data .............................................. 9-20
Determining the Class of an Object ........................................... 9-21

## Java Object Methods

Calling Syntax ............................................................................. 9-23
Obtaining Method Information .................................................. 9-25
Java Methods That Affect MATLAB Commands ......................... 9-28
How MATLAB Handles Undefined Methods ............................... 9-29
Handling Java Exceptions ........................................................... 9-30
Method Execution in MATLAB .................................................. 9-30

## Java Arrays

Introduction ..................................................................................... 9-31
How MATLAB Represents the Java Array .................................. 9-31
Creating an Array of Objects in MATLAB ................................. 9-35
Accessing Elements of a Java Array ............................................ 9-38
Assigning to a Java Array ........................................... 9-41
Concatenating Java Arrays .............................. 9-44
Creating a New Array Reference ....................... 9-45
Creating a Copy of a Java Array ...................... 9-46

Pass Data to Java Methods ................................. 9-48
  Introduction ................................................. 9-48
  Conversion of MATLAB Argument Data .............. 9-48
  Passing Built-In Types .................................. 9-50
  Converting Numbers to Integer Arguments ......... 9-51
  Passing String Arguments .............................. 9-52
  Passing Java Objects .................................... 9-52
  Other Data Conversion Topics ....................... 9-55
  Passing Data to Overloaded Methods ............... 9-56

Handle Data Returned from Java Methods ............. 9-58
  Introduction ................................................. 9-58
  Conversion of Java Return Types .................... 9-58
  Conversion of Java Object Return Types .......... 9-59
  Built-In Types ............................................. 9-59
  Converting Objects to MATLAB Types .............. 9-60

Read URL .......................................................... 9-64
  Overview ..................................................... 9-64
  Description of URLdemo ............................... 9-64
  Running the Example ................................. 9-65

Find Internet Protocol Address ......................... 9-66
  Overview ..................................................... 9-66
  Description of resolveip ............................. 9-66
  Running the Example ................................. 9-67

Create and Use Phone Book ................................. 9-68
  Overview ..................................................... 9-68
  Description of Function phonebook ............... 9-69
  Description of Function pb_lookup ............... 9-73
  Description of Function pb_add .................... 9-73
  Description of Function pb_remove ............... 9-74
  Description of Function pb_change ............... 9-75
  Description of Function pb_listall .............. 9-76
  Description of Function pb_display .............. 9-77
  Description of Function pb_keyfilter .......... 9-77
  Running the phonebook Program ................... 9-78
Using .NET Libraries from MATLAB

Read Cell Arrays of Excel Spreadsheet Data ............... 10-4

Access a Simple .NET Class .................................. 10-6
  System.DateTime Example .................................. 10-6
  Create .NET Object From Constructor .................. 10-7
  View Information About .NET Object .................. 10-7
  Introduction to .NET Data Types ..................... 10-9

Load a Global .NET Assembly ............................... 10-11

Work with Microsoft Excel Spreadsheets Using .NET ..... 10-12

Work with Microsoft Word Documents Using .NET ....... 10-14

Assembly is Library of .NET Classes ................. 10-15

Limitations to .NET Support .............................. 10-16

System Requirements for Using MATLAB Interface
to .NET ......................................................... 10-18
  MATLAB Configuration File .............................. 10-18

Using .NET from MATLAB ................................. 10-19
  Benefits of the MATLAB .NET Interface .............. 10-19
  Why Use the MATLAB .NET Interface? .............. 10-19
  NET Assembly Integration Using MATLAB Compiler SDK 10-20
  To Learn More About the .NET Framework .......... 10-20

Using a .NET Object ......................................... 10-21
  Creating a .NET Object .................................. 10-21
  What Classes Are in a .NET Assembly? ............... 10-21
  Using the delete Function on a .NET Object ........ 10-22

Build a .NET Application for MATLAB Examples ....... 10-23
Troubleshooting Security Policy Settings From Network Drives ......................................................... 10-24

.NET Terminology .......................................................................................................................... 10-25
  .NET Framework System Namespace ......................................................................................... 10-25
  Reference Type Versus Value Type ......................................................................................... 10-25

Simplify .NET Class Names ......................................................................................................... 10-26

Use import in MATLAB Functions .............................................................................................. 10-27

Nested Classes .............................................................................................................................. 10-28

Handle .NET Exceptions .............................................................................................................. 10-29

Pass Numeric Arguments ............................................................................................................ 10-30
  Call .NET Methods with Numeric Arguments .................................................................... 10-30
  Use .NET Numeric Types in MATLAB ............................................................................. 10-30

Pass System.String Arguments ............................................................................................... 10-31
  Call .NET Methods with System.String Arguments ...................................................... 10-31
  Use System.String in MATLAB ...................................................................................... 10-31

Pass System.Enum Arguments .................................................................................................. 10-33
  Call .NET Methods with System.Enum Arguments ......................................................... 10-33
  Use System.Enum in MATLAB .......................................................................................... 10-34

Pass System.Nullable Arguments .............................................................................................. 10-35

Pass Cell Arrays of .NET Data .................................................................................................... 10-39
  Example of Cell Arrays of .NET Data .............................................................................. 10-39
  Create a Cell Array for Each System.Object .................................................................. 10-40
  Create MATLAB Variables from the .NET Data ............................................................. 10-40
  Call MATLAB Functions with MATLAB Variables ...................................................... 10-40

Pass Jagged Arrays ...................................................................................................................... 10-42
  Create System.Double .NET Jagged Array ...................................................................... 10-42
  Call .NET Method with System.String Jagged Array Arguments ............................................... 10-42
  Call .NET Method with Multidimensional Jagged Array Arguments .............................................. 10-43

Convert Nested System.Object Arrays ..................................................................................... 10-45
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work with .NET Methods Having Multiple Signatures</td>
<td>10-69</td>
</tr>
<tr>
<td>Display Function Signature Example</td>
<td>10-70</td>
</tr>
<tr>
<td>Call .NET Methods With <code>out</code> Keyword</td>
<td>10-71</td>
</tr>
<tr>
<td>Call .NET Methods With <code>ref</code> Keyword</td>
<td>10-73</td>
</tr>
<tr>
<td>Call .NET Methods With <code>params</code> Keyword</td>
<td>10-75</td>
</tr>
<tr>
<td>Call .NET Methods with Optional Arguments</td>
<td>10-77</td>
</tr>
<tr>
<td>Setting Up the Examples</td>
<td>10-77</td>
</tr>
<tr>
<td>Skip Optional Arguments</td>
<td>10-77</td>
</tr>
<tr>
<td>Call Overloaded Methods</td>
<td>10-78</td>
</tr>
<tr>
<td>Calling .NET Methods</td>
<td>10-80</td>
</tr>
<tr>
<td>Calling Object Methods</td>
<td>10-80</td>
</tr>
<tr>
<td>Getting Method Information</td>
<td>10-80</td>
</tr>
<tr>
<td>C# Method Access Modifiers</td>
<td>10-80</td>
</tr>
<tr>
<td>VB.NET Method Access Modifiers</td>
<td>10-81</td>
</tr>
<tr>
<td>Reading Method Signatures</td>
<td>10-81</td>
</tr>
<tr>
<td>Call .NET Methods with Optional Arguments</td>
<td>10-83</td>
</tr>
<tr>
<td>Skipping Optional Arguments</td>
<td>10-83</td>
</tr>
<tr>
<td>Determining Which Overloaded Method Is Invoked</td>
<td>10-83</td>
</tr>
<tr>
<td>Support for <code>ByRef</code> Attribute in VB.NET</td>
<td>10-83</td>
</tr>
<tr>
<td>Calling .NET Extension Methods</td>
<td>10-84</td>
</tr>
<tr>
<td>Call .NET Properties That Take an Argument</td>
<td>10-85</td>
</tr>
<tr>
<td>How MATLAB Represents .NET Operators</td>
<td>10-87</td>
</tr>
<tr>
<td>Limitations to Support of .NET Methods</td>
<td>10-88</td>
</tr>
<tr>
<td>Overloading MATLAB Functions</td>
<td>10-88</td>
</tr>
<tr>
<td>Use .NET Events in MATLAB</td>
<td>10-89</td>
</tr>
<tr>
<td>Monitor Changes to .TXT File</td>
<td>10-89</td>
</tr>
<tr>
<td>Monitor Changes to Windows Form ComboBox</td>
<td>10-89</td>
</tr>
<tr>
<td>Call .NET Delegates in MATLAB</td>
<td>10-92</td>
</tr>
<tr>
<td>Declare a Delegate in a C# Assembly</td>
<td>10-92</td>
</tr>
<tr>
<td>Load the Assembly Containing the Delegate into MATLAB</td>
<td>10-92</td>
</tr>
<tr>
<td>Select a MATLAB Function</td>
<td>10-92</td>
</tr>
</tbody>
</table>
Create an Instance of the Delegate in MATLAB .......................... 10-93
Invoke the Delegate Instance in MATLAB ................................. 10-93

Create Delegates from .NET Object Methods ............................... 10-94

Create Delegate Instances Bound to .NET Methods ..................... 10-95
  Example — Create a Delegate Instance Associated with a .NET
  Object Instance Method .............................................. 10-95
  Example — Create a Delegate Instance Associated with a
  Static .NET Method .................................................... 10-96

Call Delegates With out and ref Type Arguments ....................... 10-97

Combine and Remove .NET Delegates ...................................... 10-98

.NET Delegates .................................................................. 10-100

Calling .NET Methods Asynchronously ..................................... 10-101
  How MATLAB Handles Asynchronous Method Calls
  in .NET ...................................................................... 10-101
  Calling a Method Asynchronously Using a Callback When an
  Asynchronous Call Finishes ............................................ 10-101
  Calling a Method Asynchronously Without a Callback ............ 10-103
  Using EndInvoke With out and ref Type Arguments .............. 10-104
  Using Polling to Detect When Asynchronous Call Finishes ....... 10-104

Limitations to Support of .NET Events ................................. 10-105
  MATLAB Support of Standard Signature of an Event Handler
  Delegate ................................................................. 10-105

Limitations to Support of .NET Delegates ............................... 10-106

Use Bit Flags with .NET Enumerations ................................. 10-107
  How MATLAB Supports Bit-Wise Operations on
  System.Enum .............................................................. 10-107
  Creating .NET Enumeration Bit Flags ................................ 10-107
  Removing a Flag from a Variable .................................... 10-108
  Replacing a Flag in a Variable ....................................... 10-108
  Testing for Membership ............................................... 10-109

Read Special System Folder Path ............................................. 10-111

.NET Enumerations in MATLAB ............................................. 10-112
Default Methods for an Enumeration ........................................... 10-113
NetDocEnum Example Assembly ................................................... 10-115
Work with Members of a .NET Enumeration ................................. 10-116
Refer to a .NET Enumeration Member ........................................ 10-118
   Using the Implicit Constructor ............................................. 10-118
Display .NET Enumeration Members as Character Strings .............. 10-120
Convert .NET Enumeration Values to Type Double ....................... 10-121
Iterate Through a .NET Enumeration ........................................ 10-122
   Information About System.Enum Methods .............................. 10-122
   Display Enumeration Member Names .................................... 10-123
Use .NET Enumerations to Test for Conditions ............................ 10-124
   Using Switch Statements .................................................. 10-124
   Using Relational Operations ............................................ 10-124
Underlying Enumeration Values ............................................... 10-126
Limitations to Support of .NET Enumerations .............................. 10-127
Create .NET Collections .......................................................... 10-128
Convert .NET Collections to MATLAB Arrays .............................. 10-130
Create .NET Arrays of Generic Type ........................................ 10-131
Display .NET Generic Methods Using Reflection ......................... 10-132
   showGenericMethods Function ......................................... 10-132
   Display Generic Methods in a Class .................................. 10-133
   Display Generic Methods in a Generic Class ....................... 10-134
.NET Generic Classes ............................................................ 10-135
Accessing Items in .NET Collections ....................................... 10-136
Call .NET Generic Methods ..................................................... 10-137
   Using the NetDocGeneric Example ..................................... 10-137
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke Generic Class Member Function</td>
<td>10-138</td>
</tr>
<tr>
<td>Invoke Static Generic Functions</td>
<td>10-138</td>
</tr>
<tr>
<td>Invoke Static Generic Functions of a Generic Class</td>
<td>10-138</td>
</tr>
<tr>
<td>Invoke Generic Functions of a Generic Class</td>
<td>10-139</td>
</tr>
<tr>
<td>Using COM Objects from MATLAB</td>
<td>11</td>
</tr>
<tr>
<td>MATLAB COM Integration</td>
<td>11-2</td>
</tr>
<tr>
<td>Concepts and Terminology</td>
<td>11-2</td>
</tr>
<tr>
<td>COM Objects, Clients, and Servers</td>
<td>11-2</td>
</tr>
<tr>
<td>Interfaces</td>
<td>11-3</td>
</tr>
<tr>
<td>The MATLAB COM Client</td>
<td>11-3</td>
</tr>
<tr>
<td>The MATLAB COM Automation Server</td>
<td>11-4</td>
</tr>
<tr>
<td>Registering Controls and Servers</td>
<td>11-5</td>
</tr>
<tr>
<td>Accessing COM Controls Created with .NET</td>
<td>11-5</td>
</tr>
<tr>
<td>Verifying the Registration</td>
<td>11-5</td>
</tr>
<tr>
<td>Getting Started with COM</td>
<td>11-7</td>
</tr>
<tr>
<td>Creating an Instance of a COM Object</td>
<td>11-7</td>
</tr>
<tr>
<td>Getting Information About a Particular COM Control</td>
<td>11-7</td>
</tr>
<tr>
<td>Getting an Object's ProgID</td>
<td>11-8</td>
</tr>
<tr>
<td>Registering a Custom Control</td>
<td>11-8</td>
</tr>
<tr>
<td>Use Internet Explorer in MATLAB Figure</td>
<td>11-10</td>
</tr>
<tr>
<td>Techniques Demonstrated</td>
<td>11-10</td>
</tr>
<tr>
<td>Using the Figure to Access Properties</td>
<td>11-10</td>
</tr>
<tr>
<td>Complete Code Listing</td>
<td>11-11</td>
</tr>
<tr>
<td>Creating the Figure</td>
<td>11-12</td>
</tr>
<tr>
<td>Calculating the ActiveX Object Container Size</td>
<td>11-12</td>
</tr>
<tr>
<td>Automatic Resize</td>
<td>11-13</td>
</tr>
<tr>
<td>Selecting Graphics Objects</td>
<td>11-13</td>
</tr>
<tr>
<td>Add Grid ActiveX Control in a Figure</td>
<td>11-15</td>
</tr>
<tr>
<td>Techniques Demonstrated</td>
<td>11-15</td>
</tr>
<tr>
<td>Using the Control</td>
<td>11-15</td>
</tr>
<tr>
<td>Complete Code Listing</td>
<td>11-16</td>
</tr>
<tr>
<td>Preparing to Use the Control</td>
<td>11-17</td>
</tr>
<tr>
<td>Creating a Figure to Contain the Control</td>
<td>11-18</td>
</tr>
</tbody>
</table>
Contents

Creating an Instance of the Control ........................................ 11-19
Using Mouse-Click Event to Plot Data ................................. 11-20
Managing Figure Resize ...................................................... 11-21
Closing the Figure ............................................................. 11-22

Read Spreadsheet Data Using Excel as Automation
Server ................................................................. 11-23
Techniques Demonstrated .................................................. 11-23
Using the UI ................................................................. 11-23
Complete Code Listing ...................................................... 11-24
Excel Spreadsheet Format .................................................. 11-24
Excel Automation Server ................................................... 11-25
Manipulating the Data in the MATLAB Workspace ............. 11-26
The Plotter UI ............................................................... 11-26
Inserting MATLAB Graphs Into Excel Spreadsheets ............ 11-28

Supported Client/Server Configurations .............................. 11-30
Introduction ................................................................. 11-30
MATLAB Client and In-Process Server ............................. 11-30
MATLAB Client and Out-of-Process Server ....................... 11-31
COM Implementations Supported by MATLAB Software .... 11-32
Client Application and MATLAB Automation Server .......... 11-32
Client Application and MATLAB Engine Server ................ 11-33

MATLAB COM Client Support

Create COM Objects ...................................................... 12-3
Creating the Server Process — An Overview ..................... 12-3
Creating an ActiveX Control .......................................... 12-4
Creating a COM Server .................................................. 12-10

Handle COM Data in MATLAB .......................................... 12-12
Passing Data to COM Objects .......................................... 12-12
Handling Data from COM Objects .................................... 12-14
Unsupported Types ......................................................... 12-15
Passing MATLAB Data to ActiveX Objects ....................... 12-15
Passing MATLAB SAFEARRAY to COM Object .................. 12-15
Reading SAFEARRAY from COM Objects in MATLAB
Applications ............................................................... 12-17
Displaying MATLAB Syntax for COM Objects ................. 12-18

**COM Object Properties** ........................................ 12-21
MATLAB Functions for Object Properties ..................... 12-21
Work with Multiple Objects ...................................... 12-21
Enumerated Values for Properties .............................. 12-22
Property Inspector .................................................... 12-22
Custom Properties ..................................................... 12-23
Properties That Take Arguments ................................. 12-23

**COM Methods** ..................................................... 12-24
Getting Method Information ....................................... 12-24
Calling Object Methods ............................................ 12-24
Specifying Enumerated Parameters ............................. 12-25
Skipping Optional Input Arguments ............................. 12-25
Returning Multiple Output Arguments ......................... 12-26

**COM Events** ..................................................... 12-27

**COM Event Handlers** .......................................... 12-29
Overview of Event Handling ..................................... 12-29
Arguments Passed to Event Handlers .......................... 12-29
Event Structure ....................................................... 12-30

**COM Object Interfaces** ....................................... 12-32
IUnknown and IDispatch Interfaces ............................ 12-32
Custom Interfaces .................................................... 12-33

**Save and Delete COM Objects** ................................. 12-35
Functions for Saving and Restoring COM Objects ............ 12-35
Releasing COM Interfaces and Objects ....................... 12-35

**MATLAB Application as DCOM Client** .......................... 12-37

**Explore COM Objects** .......................................... 12-38
Exploring Properties ............................................... 12-38
Exploring Methods ................................................... 12-39
Exploring Events ..................................................... 12-39
Exploring Interfaces ............................................... 12-40
Identifying Objects and Interfaces ............................ 12-40

**Change Row Height in Range of Spreadsheet Cells** ...... 12-42
Write Data to Excel Spreadsheet Using ActiveX .......... 12-44
Change Cursor in Spreadsheet .......................... 12-46
Insert Spreadsheet After First Sheet .................. 12-47
Redraw Circle in mwsamp Control .................... 12-48
Connect to Existing Excel Application ................. 12-50
Display Message for Workbook OnClose Event ....... 12-51
Run Macro in Excel Server Application ............... 12-52
Combine Event Handlers as MATLAB Local Functions .. 12-53
MATLAB Sample ActiveX Control mwsamp ............. 12-54
Display Event Messages from mwsamp Control ........ 12-55
Add Position Property to mwsamp Control ............ 12-58
Save mwsamp2 COM Control ........................... 12-59
Deploy ActiveX Controls Requiring Run-Time Licenses .. 12-60
    Create a Function to Build the Control .......... 12-60
    Build the Control and the License File ........ 12-60
    Build the Executable ................................ 12-61
    Deploy the Files .................................. 12-61
Microsoft Forms 2.0 Controls .......................... 12-62
    Affected Controls ............................... 12-62
    Replacement Controls .......................... 12-62
COM Collections ..................................... 12-64
MATLAB COM Support Limitations ...................... 12-65
Interpreting Argument Callouts in COM Error Messages . 12-66
Using Web Services with MATLAB

14

Set Up WSDL Tools ........................................ 14-2

Display a World Map ...................................... 14-3

Using WSDL Web Service with MATLAB ............. 14-8
  What Are Web Services in MATLAB? ..................... 14-8
  What are WSDL Documents? ............................... 14-8
  What You Need to Use WSDL with MATLAB .......... 14-9

Access Services That Use WSDL Documents .......... 14-10

Error Handling ........................................... 14-12
  Considerations Using Web Services ................. 14-12
  Error Handling with try/catch Statements  .......... 14-12
  Use a Local Copy of the WSDL Document .......... 14-12
  Java Errors Accessing Service ....................... 14-13
  Anonymous Types Not Supported .................... 14-13

XML-MATLAB Data Type Conversion .................... 14-14

Limitations to WSDL Document Support ................ 14-16
System Commands

15

Shell Escape Functions ........................................... 15-2
Run External Commands, Scripts, and Programs .......... 15-3
  Run UNIX Programs off System Path ....................... 15-4
Change Environment Variable for Shell Command ........ 15-6

Serial Port I/O

16

Capabilities and Supported Interfaces and Platforms .... 16-2
  What Is the MATLAB Serial Port Interface? .............. 16-2
  Supported Serial Port Interface Standards ............... 16-3
  Supported Platforms ......................................... 16-3
  Using the Examples with Your Device .................... 16-3

Overview of the Serial Port ..................................... 16-4
  Introduction .................................................. 16-4
  What Is Serial Communication? ............................ 16-4
  The Serial Port Interface Standard ....................... 16-4
  Connecting Two Devices with a Serial Cable ............ 16-5
  Serial Port Signals and Pin Assignments ................. 16-6
  Serial Data Format .......................................... 16-10
  Finding Serial Port Information for Your Platform .... 16-13
  Using Virtual USB Serial Ports ............................ 16-15
  Selected Bibliography ....................................... 16-15

Getting Started with Serial I/O ............................... 16-16
  Example: Getting Started ................................... 16-16
  The Serial Port Session ...................................... 16-16
  Configuring and Returning Properties ..................... 16-17

Create a Serial Port Object .................................... 16-22
  Overview of a Serial Port Object .......................... 16-22
  Configuring Properties During Object Creation .......... 16-23
  The Serial Port Object Display ............................ 16-24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an Array of Serial Port Objects</td>
<td>16-24</td>
</tr>
<tr>
<td>Connect to the Device</td>
<td>16-26</td>
</tr>
<tr>
<td>Configure Communication Settings</td>
<td>16-27</td>
</tr>
<tr>
<td>Write and Read Data</td>
<td>16-28</td>
</tr>
<tr>
<td>Before You Begin</td>
<td>16-28</td>
</tr>
<tr>
<td>Example — Introduction to Writing and Reading Data</td>
<td>16-28</td>
</tr>
<tr>
<td>Controlling Access to the MATLAB Command Line</td>
<td>16-29</td>
</tr>
<tr>
<td>Writing Data</td>
<td>16-30</td>
</tr>
<tr>
<td>Reading Data</td>
<td>16-35</td>
</tr>
<tr>
<td>Example — Writing and Reading Text Data</td>
<td>16-41</td>
</tr>
<tr>
<td>Example — Parsing Input Data Using textscan</td>
<td>16-42</td>
</tr>
<tr>
<td>Example — Reading Binary Data</td>
<td>16-43</td>
</tr>
<tr>
<td>Events and Callbacks</td>
<td>16-46</td>
</tr>
<tr>
<td>Introduction</td>
<td>16-46</td>
</tr>
<tr>
<td>Example — Introduction to Events and Callbacks</td>
<td>16-46</td>
</tr>
<tr>
<td>Event Types and Callback Properties</td>
<td>16-47</td>
</tr>
<tr>
<td>Respond To Event Information</td>
<td>16-49</td>
</tr>
<tr>
<td>Create and Execute Callback Functions</td>
<td>16-51</td>
</tr>
<tr>
<td>Enable Callback Functions After They Error</td>
<td>16-52</td>
</tr>
<tr>
<td>Example — Using Events and Callbacks</td>
<td>16-52</td>
</tr>
<tr>
<td>Control Pins</td>
<td>16-54</td>
</tr>
<tr>
<td>Properties of Serial Port Control Pins</td>
<td>16-54</td>
</tr>
<tr>
<td>Signaling the Presence of Connected Devices</td>
<td>16-54</td>
</tr>
<tr>
<td>Controlling the Flow of Data: Handshaking</td>
<td>16-57</td>
</tr>
<tr>
<td>Debugging: Recording Information to Disk</td>
<td>16-60</td>
</tr>
<tr>
<td>Introduction</td>
<td>16-60</td>
</tr>
<tr>
<td>Recording Properties</td>
<td>16-60</td>
</tr>
<tr>
<td>Example: Introduction to Recording Information</td>
<td>16-61</td>
</tr>
<tr>
<td>Creating Multiple Record Files</td>
<td>16-61</td>
</tr>
<tr>
<td>Specifying a Filename</td>
<td>16-61</td>
</tr>
<tr>
<td>The Record File Format</td>
<td>16-62</td>
</tr>
<tr>
<td>Example: Recording Information to Disk</td>
<td>16-63</td>
</tr>
<tr>
<td>Save and Load</td>
<td>16-66</td>
</tr>
<tr>
<td>Using save and load</td>
<td>16-66</td>
</tr>
<tr>
<td>Using Serial Port Objects on Different Platforms</td>
<td>16-66</td>
</tr>
</tbody>
</table>
### Disconnect and Clean Up

- Disconnect a Serial Port Object: 16-68
- Clean Up the MATLAB Environment: 16-68

### Property Reference

- The Property Reference Page Format: 16-70
- Serial Port Object Properties: 16-70

### Properties — Alphabetical List

### Hardware Support

17

### Support Package Installation

- What Is a Support Package?: 17-2
- Install Support Packages: 17-2
- Install Downloaded Support Package on Multiple Computers: 17-3
- Troubleshoot Timed Out Connections: 17-4

### Support Package Installer Help

- About Support Package Installer: 17-5
- Select an Action: 17-6
- Select Support Package to Install or Select Support Package to Download: 17-7
- Log In to MathWorks Account: 17-10
- The MathWorks, Inc. Software License Agreement: 17-12
- Third-Party Software Licenses: 17-13
- Confirm Installation, Confirm Download, Confirm Uninstall: 17-14
- Install or Update Complete, Download Complete, Uninstall Complete: 17-15
- Set Up Support Package: 17-16
- Support Package Setup Complete: 17-18
- Special Instructions: 17-18

### MATLAB Supported Hardware

17-19
Python Interface Topics

• “Install Supported Python Implementation” on page 1-3
• “Call Python from MATLAB” on page 1-4
• “Call User-Defined Python Module” on page 1-10
• “Use Python Numeric Types in MATLAB” on page 1-12
• “Call Python Methods with Numeric Arguments” on page 1-13
• “Default Numeric Types” on page 1-14
• “Use Python array Types in MATLAB” on page 1-15
• “Pass MATLAB String to Python Method” on page 1-16
• “Use Python str Type in MATLAB” on page 1-17
• “Pass MATLAB Backslash Control Character” on page 1-18
• “Create Python list Variable” on page 1-19
• “Use Python list Type in MATLAB” on page 1-20
• “Use Python List of Numeric Types in MATLAB” on page 1-22
• “Pass Cell Array as Python Sequence Type” on page 1-23
• “Read Element of Nested list Type” on page 1-25
• “Use Python tuple Type in MATLAB” on page 1-26
• “Create Python tuple Variable” on page 1-27
• “Create Singleton Python tuple Variable” on page 1-28
• “Create Python dict Variable” on page 1-29
• “Pass dict Argument to Python Method” on page 1-30
• “Use Python dict Type in MATLAB” on page 1-31
• “Convert Python dict Type to MATLAB Structure” on page 1-33
• “Pass Keyword Arguments” on page 1-34
• “Pass Python Function to Python map Function” on page 1-35
• “Index into Python String” on page 1-36
• “Index into Python List” on page 1-37
• “Index into Python Tuple” on page 1-39
• “Index into Python dict” on page 1-40
• “Use Python List as Values in for Loop” on page 1-41
• “Display Stepped Range of Elements” on page 1-42
• “Access Elements in Python Container Types” on page 1-43
• “View Python Numeric Values” on page 1-46
• “Call Methods on Python Variables” on page 1-48
• “Reload Modified User-Defined Python Module” on page 1-50
• “System and Configuration Requirements” on page 1-52
• “Create a Python Object” on page 1-54
• “Pass Data to Python” on page 1-57
• “Handle Data Returned from Python” on page 1-60
• “How MATLAB Represents Python Operators” on page 1-62
• “Execute Callable Python Object” on page 1-64
• “Python import and MATLAB import Commands” on page 1-65
• “List, Tuple, and Dictionary Types” on page 1-67
• “Limitations to Python Support” on page 1-68
• “Limitations to Indexing into Python Objects” on page 1-69
• “Undefined variable "py" or function "py.command"” on page 1-70
• “Help for Python Functions” on page 1-73
• “Handle Python Exceptions” on page 1-74
• “Troubleshooting Error Messages” on page 1-75
• “Using Python Data in MATLAB” on page 1-77
• “Call Python eval Function” on page 1-78
• “Precedence Order of Methods and Functions” on page 1-80
• “Python Function Arguments” on page 1-81
Install Supported Python Implementation

In this section...

“Install Python Version 2.7” on page 1-3
“Install Python Version 3.3 or 3.4” on page 1-3

Install Python Version 2.7

• Access https://www.python.org/downloads and scroll down to the Looking for a specific release section.
• Find the 2.7 version you want and click Download.
• Click the format you want, and follow the online instructions.

Note: For 64-bit MATLAB® on Microsoft® Windows® systems, select the 64-bit Python® version, called Windows x86-64 MSI installer.

Install Python Version 3.3 or 3.4

• Access https://www.python.org/downloads and scroll down to the Looking for a specific release section.
• Find the 3.3/3.4 version you want and click Download.
• Click the format you want, and follow the online instructions.

Note: For 64-bit MATLAB on Windows systems, select the 64-bit Python version, called Windows x86-64 MSI installer.

See Also
pyversion

More About
• “MATLAB Cannot Find Python” on page 1-71
Call Python from MATLAB

These examples show how to use Python® language functions and modules within MATLAB®. The first example calls a text-formatting module from the Python standard library. The second example shows how to use a third-party module, Beautiful Soup. If you want to run that example, follow the guidelines in the step for installing the module.

MATLAB supports the reference implementation of Python, often called CPython, versions 2.7, 3.3, and 3.4. 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”.

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.

\[
T = 'MATLAB(R) is a high-level language and interactive environment for numerical computation, visualization, and reach a solution faster than with spreadsheets or traditional programming languages, such as C/C++ or Java(TM).';
\]

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               112  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.

\[
wrapped = cell(wrapped);
\]

\[
whos wrapped
\]
Although `wrapped` is a MATLAB cell array, each cell element is a Python string.

```matlab
cellfun(@char, wrapped, 'UniformOutput', false);
```

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.

```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
', 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).

**Use Beautiful Soup, a Third-Party Python Module**

This example shows how to use a third-party module, Beautiful Soup, a tool for parsing HTML. If you want to run the example, you need to install this module using apt-get, pip, easy_install, or other tool you use to install Python modules.

First, find a Web page that includes a table of data. This example uses a table of the population of the world from the following English-language Wikipedia site. This example assumes the first table contains the population data, and assumes the country name is in the second column and the third column contains the population.

```python
soup = py.bs4.BeautifulSoup(html);
```

Next, extract all of the table data from the HTML, creating a cell array. If you want a deeper understanding of what is happening, refer to the documentation for Beautiful Soup.
tables = soup.find_all('table');
t = cell(tables);

The first table is the one of interest; extract its rows.

c = cell(t{1}.find_all('tr'));
c = cell(c)

Now loop over the cell array, extracting the country name and population from each row, found in the second and third columns respectively.

countries = cell(size(c));
populations = nan(size(c));

for i = 1:numel(c)
    row = c{i};
    row = cell(row.find_all('td'));
    if ~isempty(row)
        countries{i} = char(row{2}.get_text());
        populations(i) = str2double(char(row{3}.get_text()));
    end
end

Finally, create a MATLAB table from the data, and eliminate any lingering nan values; these NaNs represented invalid rows when importing the HTML.

data = table(countries, populations, ...
    'VariableNames', {'Country', 'Population'});
data = data(~isnan(data.Population), :);

Trim the tail end of the table and make a pie chart

restofWorldPopulation = sum(data.Population(11:end));
data = data(1:10, :);
data = [data;table({' Rest of World'}, restofWorldPopulation, ...
    'VariableNames', {'Country', 'Population'})]
pie(data.Population)
legend(data.Country, 'Location', 'EastOutside');
title('Distribution of World Population')

data =

    Country       Population

1-7
<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>China[8]</td>
<td>1.3679e+09</td>
</tr>
<tr>
<td>India</td>
<td>1.2624e+09</td>
</tr>
<tr>
<td>United States</td>
<td>3.1908e+08</td>
</tr>
<tr>
<td>Indonesia</td>
<td>2.5216e+08</td>
</tr>
<tr>
<td>Brazil</td>
<td>2.0344e+08</td>
</tr>
<tr>
<td>Pakistan</td>
<td>1.8813e+08</td>
</tr>
<tr>
<td>Nigeria</td>
<td>1.7852e+08</td>
</tr>
<tr>
<td>Bangladesh</td>
<td>1.5731e+08</td>
</tr>
<tr>
<td>Russia[9]</td>
<td>1.4615e+08</td>
</tr>
<tr>
<td>Japan</td>
<td>1.2709e+08</td>
</tr>
<tr>
<td>Rest of World</td>
<td>2.9697e+09</td>
</tr>
</tbody>
</table>

Distribution of World Population

- China[8]: 19%
- India: 18%
- United States: 4%
- Indonesia: 4%
- Brazil: 4%
- Pakistan: 3%
- Nigeria: 3%
- Bangladesh: 2%
- Russia[9]: 2%
- Japan: 2%
- Rest of World: 41%
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 “Call Python Libraries”.
Call User-Defined Python Module

This example shows how to call methods from the following Python module. This module contains functions used by examples in this documentation.

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.

```python
# mymod.py
"""Python module demonstrates passing MATLAB types to Python functions""

# 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.

```matlab
if count(py.sys.path,'') == 0
    insert(py.sys.path,int32(0),'');
end
```

In the mymod.py source file, read the function signature for the search function. The function takes one input argument, words.

```python
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:

```python
"""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']

Related Examples
•  “Reload Modified User-Defined Python Module” on page 1-50

External Websites
•  PEP 257 -- Docstring Conventions
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. MATLAB automatically converts this type to double.

```python
pynum = py.math.radians(90)
class(pynum)

pynum =
    1.5708

ans =
double
```

More About
- “Pass Data to Python” on page 1-57
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` sample data file and read the numeric array, `Height`.

```matlab
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.

```matlab
size(Height)
ans =
    100     1

Transform `Height` to a 1-by-N matrix before calling `fsum`.

```matlab
py.math.fsum(Height')
ans =
    6707
```

**More About**

- “Pass Data to Python” on page 1-57
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.

```python
d = py.datetime.date(2014,12,31)
```

Python Error: TypeError: integer argument expected, got float

Explicitly convert each number to an integer type:

```python
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.

```python
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.

```matlab
A = double(P);
```

Sum the elements of A.

```matlab
sum(A)
```

```matlab
ans =

15
```
Pass MATLAB String 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 string representing a valid folder.

```matlab
folder = fullfile(matlabroot,'help','examples');
```

Pass the string to the `os.listdir` function. MATLAB automatically converts `folder` to the Python `str` type.

```matlab
F = py.os.listdir(folder)
```

```python
F =

Python list with no properties.

['graphics', 'graphics2', 'matlab']
```

MATLAB displays a list of folders, based on your product.

More About

• “Pass Data to Python” on page 1-57
Use Python str Type in MATLAB

This example shows how to use the Python path separator character (;).

```python
p = py.os.path.pathsep
p = Python str with no properties.
;
```

In MATLAB, a Python character is a `py.str` variable.

MATLAB uses the same path separator character, ;.

```matlab
c = pathsep
c = 
;
```

Compare the MATLAB variable type to the Python type.

```matlab
isequal(class(p),class(c))
an =

0
```

A `py.str` type is not equivalent to a MATLAB `char` type.

Convert p to a MATLAB type and append the character to a file name.

```matlab
f = ['myfile' char(p)]
f = myfile;
```

More About

- “Explicit Type Conversions” on page 1-60
Pass MATLAB Backslash Control Character

This example shows how to pass the backslash control character (\) to a Python str type.

Insert the new line control character, \n, by calling the MATLAB sprintf function.

```python
py.str(sprintf('The rain\n\nin Spain.'))
```

```plaintext
ans =
    Python str with no properties.
        The rain
        in Spain.
```

Python replaces \n with a new line.

Without the sprintf function, both MATLAB and Python interpret \ as a literal backslash.

```python
py.str('The rain\n\nin Spain. ')
```

```plaintext
ans =
    Python str with no properties.
        The rain\nin Spain.
```

Pass this string to a Python string method, split.

```python
split(py.str('The rain\n\nin Spain. '))
```

```plaintext
ans =
    Python list with no properties.
        ['The', 'rain\nin', 'Spain. ']
```

Python treats the MATLAB string as a raw string and adds a \ character to preserve the original backslash.
Create Python list Variable

This example shows how to create a list variable to pass to a Python function.

```python
students = py.list({'Robert', 'Mary', 'Joseph'})
```

students =

Python list with no properties.

['Robert', 'Mary', 'Joseph']

Display number of students in the list.

```python
n = py.len(students)
```

n =

3

More About

- “List, Tuple, and Dictionary Types” on page 1-67
Use Python list Type in MATLAB

This example shows how to display folders on your system using the MATLAB \texttt{disp} function on the Python \texttt{sys.path} variable. \texttt{sys.path} is a list type.

Create a Python list, \texttt{P}, of folders on the Python search path.

\begin{verbatim}
P = py.sys.path;
class(P)
\end{verbatim}

\begin{verbatim}
an\ =
py.list
\end{verbatim}

Convert list \texttt{P} to a MATLAB type. The first step is to convert the list to a cell array of folder names, \texttt{cP}.

\begin{verbatim}
cP = cell(P);
class(cP)
\end{verbatim}

\begin{verbatim}
an\ =
cell
\end{verbatim}

Each folder name is a Python string.

\begin{verbatim}
class(cP{1})
an\ =
py.str
\end{verbatim}

Convert the Python strings in \texttt{cP} to MATLAB strings using the \texttt{char} function. Put the values into a new cell array, \texttt{cellP}. Display the folder names.

\begin{verbatim}
cellP = cell(1, numel(cP));
for n=1:numel(cP)
    strP = char(cP{n});
    cellP(n) = {strP};
    disp(strP)
end
\end{verbatim}

\begin{verbatim}
C:\Python27\lib\site-packages\protobuf-2.5.0-py2.7.egg
C:\windows\system32\python27.zip
C:\Python27\DLLs
\end{verbatim}
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**

- “Handle Data Returned from Python” on page 1-60
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 all elements of the Python list are numeric.

Suppose that you have a Python function that returns the following list of integers, `P`.

```python
P =
    Python list with no properties.
    [1, 2, 3, 4]
```

Display the numeric type of the values.

```matlab
class(P{1})
an =
int64
```

Convert `P` to a MATLAB cell array.

```matlab
cP = cell(P);
```

Convert the cell array to a MATLAB array of `double`.

```matlab
A = cellfun(@double,cP)
A =
    1  2  3  4
```

See Also
- `cell`
- `cellfun`

Related Examples
- “Use Python array Types in MATLAB” on page 1-15
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 a string 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.

```python
# 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.

```python
P = py.sys.path;
append(P,prefdir);
```

Open the MATLAB sample data file, creating the cell array, LastName.

```python
load patients.mat
```

Convert the cell array to a 1-by-N py.list array.

```python
L = py.list(LastName');
class(L)
an =
py.list
```

Call mymod.theend on the list.

```python
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.

**Related Examples**

- “Use Python list Type in MATLAB” on page 1-20
Read Element of Nested list Type

This example shows how to access an element of a Python list containing list elements.

```python
matrix = py.list({{1, 2, 3, 4}, {'hello', 'world'}, {9, 10}});
```

Display element 'world', at index (2,2).

```python
disp(char(matrix{2}{2}))
```

`world`

More About

- “Multilevel Indexing to Access Parts of Cells”
Use Python tuple Type in MATLAB

This example shows how to use a Python tuple, returned by the `os.path.split` function, in MATLAB.

```python
pn = py.os.path.split('C:\Program Files\MATLAB\R2014a\help\examples')
pn =

    Python tuple with no properties.
    ('C:\\Program Files\\MATLAB\\R2014a\\help', 'examples')
```

Convert the parts of the folder to MATLAB strings and display the results.

```matlab
head = char(pn{1})
tail = char(pn{end})

head =
C:\Program Files\MATLAB\R2014a\help

tail =
examples
```
Create Python tuple Variable

This example shows how to create a tuple variable to pass to a Python function.

```python
student = py.tuple({'Robert',19,'Biology'})

student =

Python tuple with no properties.

('Robert', 19.0, 'Biology')
```

More About

- “List, Tuple, and Dictionary Types” on page 1-67
Create Singleton Python tuple Variable

This example shows how to create a tuple variable with a single element.

```
subject = py.tuple({'Biology'})
subject =
    Python tuple with no properties.
    ('Biology',)
```

A tuple with one element has a trailing comma.
Create Python dict Variable

This example shows how to create a dict variable to pass to a Python function.

\[
\text{studentID} = \text{py.dict(pyargs('Robert',357,'Mary',229,'Jack',391))}
\]

\[
\text{studentID} = \text{Python dict with no properties.}
\]

\[
\{\text{Robert}: 357.0, \text{Jack}: 391.0, \text{Mary}: 229.0\}
\]

More About

- “List, Tuple, and Dictionary Types” on page 1-67
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.

```python
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.

```python
update(menu, py.dict(pyargs('bread', 2.50)))
menu
```

```python
menu =

     Python dict with no properties.

     {'soup': 3.57, 'bacon': 3.91, 'bread': 2.5, 'salad': 5.0}
```
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, `myfunc`, that returns menu items and prices in a dictionary, `dict` type.

```python
order = myfunc;
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. Use the Python `keys` function to display the menu items.

```python
keys(order)
ans =
    Python list with no properties.
    ['soup', 'bacon', 'salad', 'bread']
```

Use the Python `values` function to display the prices.

```python
values(order)
ans =
    Python list with no properties.
    [3.57, 3.91, 5.0, 2.29]
```

Use the `cell` function to convert the Python list to a MATLAB variable.

```python
p = cell(values(order))
p =
    [3.5700]    [3.9100]    [5]    [2.2900]
```
Convert the prices to a MATLAB array.

```matlab
prices = cellfun(@double,p)
```

```
prices =

    3.57    3.91    5.00    2.29
```

Calculate the total.

```matlab
total = sum(prices)
```

```
total =

    14.77
```
**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.

```python
patient
```

```python
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.

```matlab
P = struct(patient)
```

```matlab
P =
```

```matlab
    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.

```matlab
bar([double(P.test1);double(P.test2);double(P.test3)])
```

MATLAB displays a bar graph.
Pass Keyword Arguments

The Python built-in `print` function has keyword arguments, `sep`, `end`, and `file`.

```python
print(*objects, sep=' ', end='
', file=sys.stdout)
```

The following examples use the default value for `file`.

Create some string variables.

```python
x1 = py.str('c:');
x2 = py.os.getcwd();
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.

```python
py.print(x1,x2,x3,pyargs('sep',sprintf('
')))
c: .
None
```

Use the following statement to change `sep` to an empty string and change the `end` value to display THE END.

```python
py.print(x1,x2,x3,pyargs('end', sprintf(' THE END\n'),'sep',py.str))
c:.None THE END
```

See Also

`pyargs`

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'});

Apply the Python `len` function to the `py.map` function to display the length of each word. Use the MATLAB function handle notation, `@`, to indicate `py.len` is a function.

`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]
```

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 string with indexing into the Python variable.

Create a MATLAB string and display the first character.

```matlab
str = 'myfile';
str(1)

ans =

m
```

Convert the string to a Python `str` type and display the first character.

```python
pstr = py.str(str);
pstr(1)

ans =

Python str with no properties.

m
```
This example shows how to display the last element in a Python list variable. The example compares indexing into a MATLAB cell array with indexing into a Python list.

Create a MATLAB cell array and display the last element.

```matlab
C = {1,2,3,4};
n = C(end)
n =
   [4]
```
MATLAB returns a cell array.

Display the contents of the last element.

```matlab
n = C{end}
n =
   4
```
Convert the cell array to a Python list.

```matlab
li = py.list(C)
li =
   Python list with no properties.
   [1.0, 2.0, 3.0, 4.0]
```
Display the last element.

```matlab
n = li(end)
n =
   Python list with no properties.
   [4.0]
```
MATLAB returns a list.
Display the contents of the last element.

```python
n = li{end}

n =

4
```
Index into Python Tuple

This example shows how to display elements in a tuple.

Create a tuple and display the first two elements.

t = py.tuple({'a','bc',1,2,'def'});
t(1:2)

ans =

Python tuple with no properties.

('a', 'bc')

MATLAB returns a tuple.
Index into Python dict

This example shows how to display an element in a dictionary.

Create a dictionary variable.

```python
customers = py.dict
customers =

   Python dict with no properties.

{}
```

Populate the `dict` with customer names and account numbers.

```python
customers['Smith'] = int32(2112); 
customers['Anderson'] = int32(3010); 
customers['Audrey'] = int32(4444); 
customers['Megan'] = int32(5000); 
customers =

   Python dict with no properties.

   {'Audrey': 4444, 'Anderson': 3010, 'Smith': 2112, 'Megan': 5000}
```

The output depends on your Python version.

Read the account number for customer Anderson.

```python
acct = customers['Anderson']
acct =

   3010
```

The result is a double.

```python
class(acct)
ans =
double
```
Use Python List as Values in for Loop

This example shows how to display elements of a Python list.

```python
li = py.list({1,2,3,4});
for n = li
    disp(n{1})
end

1
2
3
4
```

Variable `n` is a `py.list` with one element.

More About

- “Access Elements in Python Container Types” on page 1-43
Display Stepped Range of 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`.

```python
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 1-43
“Mapping Types” on page 1-44
“Size and Dimensions” on page 1-44
“Array Support” on page 1-45
“Use Zero-Based Indexing for Python Functions” on page 1-45

Typically, to work with a Python variable in MATLAB, you convert the Python object to a MATLAB array, and then index into the array as needed. Sometimes, you want to preserve the Python object.

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.

**Sequence Types**

Python sequence types behave like MATLAB cell arrays.

Get a subsequence using smooth-parenthesis () indexing.

\[ li = \text{py.list}([1,2,3,4]); \]
\[ \text{res} = li(2:3) \]

\[ \text{res} = \]

Python list with no properties.

\[ [2.0, 3.0] \]

Use curly braces {} to get the contents of the element.

\[ \text{res} = li\{1\} \]

\[ \text{res} = 1 \]
**Mapping Types**

For mapping types, use curly braces with the Python `key` argument.

```python
patient = py.dict(pyargs('name','John Doe','billing',127));
patient['billing']
```

```plaintext
ans =
127
```

**Size and Dimensions**

MATLAB displays information for your system.

```python
p = py.sys.path;
class(p)
```

```plaintext
ans =
py.list
```

Index into `p`.

```python
p(1)
p{1}
```

```plaintext
ans =

Python list with no properties.

['c:\\work']
```

```plaintext
ans =

Python str with no properties.

c:\\work
```

Inspect dimensions.

```python
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 \( \text{int32}(0) \). The index value for the last element is \( \text{int32}(\text{py.len}(x) - 1) \).

**Related Examples**

- “Index into Python String” on page 1-36
- “Index into Python List” on page 1-37
- “Index into Python Tuple” on page 1-39
- “Index into Python dict” on page 1-40

**More About**

- “Explicit Type Conversions” on page 1-60
- “Limitations to Indexing into Python Objects” on page 1-69
- “Pass MATLAB Backslash Control Character” on page 1-18
View Python Numeric Values

**In this section...**

“Why Do I See Properties When I Display a Number?” on page 1-46
“What Is the L Character Attached to a Number?” on page 1-46

**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.

```python
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:

```python
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:

```python
py.long(5) + py.long(2)
```

```python
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.

```python
P = py.sys.path;
```

Display the Python functions for a list type.

```python
methods(P)
```

Methods for class `py.list`:

<table>
<thead>
<tr>
<th>append</th>
<th>count</th>
<th>display</th>
<th>ge</th>
<th>insert</th>
<th>lt</th>
<th>plus</th>
<th>reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>details</td>
<td>eq</td>
<td>gt</td>
<td>le</td>
<td>mtimes</td>
<td>pop</td>
<td>sort</td>
</tr>
<tr>
<td>char</td>
<td>disp</td>
<td>extend</td>
<td>index</td>
<td>list</td>
<td>ne</td>
<td>remove</td>
<td></td>
</tr>
</tbody>
</table>

Methods of `py.list` inherited from `handle`.

Read the documentation for `append`.

```python
py.help('list.append')
```

Help on method_descriptor in list:

```python
list.append = append(...)  
    L.append(object) -- append object to end
```

Add the current folder to the end of the path.

```python
append(P,pwd)
```

Display number of folders on the path. The list has `py.len` elements.

```python
py.len(P)
```

```python
ans =

Python int with properties:

    real: [1x1 py.int]  
    denominator: [1x1 py.int]  
    imag: [1x1 py.int]
```
numerator: [1x1 py.int]

11

11 is the number of folders on this path. Your value might be different. The type of this number is py.int.

**Related Examples**

- “Use Python list Type in MATLAB” on page 1-20
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 `pyversion` 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`.

```python
def myfunc():
    """Display message."""
    return 'version 1'
```

Call `myfunc`.

```python
py.mymod.myfunc
```

ans =

```
Python str with no properties.
version 1
```

**Modify Module**

Modify the function, replacing the `return` statement with the following:

```python
    return 'version 2'
```

Save the file.

**Unload Module**

```python
clear classes
```

MATLAB deletes all variables, scripts, and classes in the workspace.
Import Modified Module

```python
mod = py.importlib.import_module('mymod');
```

Reload Module in Python Version 2.7

```python
py.reload(mod);
```

Reload Module in Python Version 3.3

```python
py.imp.reload(mod);
```

Reload Module in Python Version 3.4

```python
py.importlib.reload(mod);
```

Call Function in Updated Module

Call the updated `myfunc` function.

```python
py.mymod.myfunc
```

```python
ans =

    Python str with no properties.
    version 2
```

See Also

clear | pyversion

Related Examples

- “Call User-Defined Python Module” on page 1-10
System and Configuration Requirements

In this section...

“Python Version Support” on page 1-52
“Set Python Version on Windows Platform” on page 1-52
“Set Python Version on Mac and Linux Platforms” on page 1-53
“64-bit/32-bit Versions of Python on Windows Platforms” on page 1-53
“Requirements for Building Python Executable” on page 1-53

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 the following versions:

• Version 2.7
• Version 3.3
• Version 3.4

To determine if your system has a supported version, use the `pyversion` function. The value set by `pyversion` is persistent across MATLAB sessions.

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 `pyversion` with the new version information.

Set Python Version on Windows Platform

On Windows platforms, use either:

`pyversion version`

or
System and Configuration Requirements

pyversion executable

**Note**: If you downloaded a Python interpreter, but did not register it in the Windows registry, use:

pyversion executable

### Set Python Version on Mac and Linux Platforms

To set the version, type:

pyversion executable

where *executable* is the full path to the Python executable file.

### 64-bit/32-bit Versions of Python on Windows Platforms

The architecture of Python must match the architecture of MATLAB. If you run a 64-bit version of MATLAB, download a 64-bit version of Python. If you run a 32-bit version of MATLAB, download a 32-bit version of Python.

**Note**: 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**

pyversion

### External Websites

- https://www.python.org/downloads
Create a Python Object

The syntax to create a Python object `pyObj` is:

```python
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__`.

```python
py.help('textwrap.TextWrapper.__init__')
```

Help on method `__init__` in textwrap.TextWrapper:

```python
textwrap.TextWrapper.__init__ = __init__(self, width=70, initial_indent='', subsequent_indent='', expand_tabs=True, ... break_long_words=True, drop_whitespace=True, break_on_hyphens=True) unbound textwrap.TextWrapper method
```

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.

```python
tw = py.textwrap.TextWrapper;
```

```python
tw =
```

<table>
<thead>
<tr>
<th>Python TextWrapper with properties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>width: 70</td>
</tr>
<tr>
<td>subsequent_indent: [1x1 py.str]</td>
</tr>
<tr>
<td>wordsep_simple_re_uni: [1x1 py._sre.SRE_Pattern]</td>
</tr>
<tr>
<td>fix_sentence_endings: 0</td>
</tr>
<tr>
<td>break_on_hyphens: 1</td>
</tr>
<tr>
<td>break_long_words: 1</td>
</tr>
<tr>
<td>wordsep_re_uni: [1x1 py._sre.SRE_Pattern]</td>
</tr>
<tr>
<td>initial_indent: [1x1 py.str]</td>
</tr>
<tr>
<td>expand_tabs: 1</td>
</tr>
</tbody>
</table>
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 string, T.

T = 'MATLAB® is a high-level language and interactive environment for numerical computation.'
w = wrap(tw,T);
whos w
Convert the `py.list` to a cell array and display the results.

```matlab
wrapped = cellfun(@char, cell(w), 'UniformOutput', false);
fprintf('%s
', 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.

**Related Examples**
- “Call Python from MATLAB” on page 1-4
- “Pass Keyword Arguments” on page 1-34

**More About**
- “MATLAB Objects”
Pass Data to Python

In this section...

“MATLAB Type to Python Type Mapping” on page 1-57
“MATLAB Vector to Python Mapping” on page 1-58
“Unsupported MATLAB Types” on page 1-58

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.

<table>
<thead>
<tr>
<th>MATLAB Input Argument Type — Scalar Values Only</th>
<th>Resulting Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>single</td>
<td></td>
</tr>
<tr>
<td>Complex single</td>
<td>complex</td>
</tr>
<tr>
<td>Complex double</td>
<td></td>
</tr>
<tr>
<td>int8</td>
<td>int</td>
</tr>
<tr>
<td>uint8</td>
<td></td>
</tr>
<tr>
<td>int16</td>
<td></td>
</tr>
<tr>
<td>uint16</td>
<td></td>
</tr>
<tr>
<td>int32</td>
<td></td>
</tr>
<tr>
<td>uint32</td>
<td>int</td>
</tr>
<tr>
<td>int64</td>
<td>long (version 2.7 only)</td>
</tr>
<tr>
<td>uint64</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>float(nan)</td>
</tr>
<tr>
<td>Inf</td>
<td>float(inf)</td>
</tr>
<tr>
<td>logical</td>
<td>bool</td>
</tr>
<tr>
<td>Structure</td>
<td>dict</td>
</tr>
<tr>
<td>Python object — py.type</td>
<td>type</td>
</tr>
<tr>
<td>function handle @py.module.function, to Python functions only</td>
<td>module.function</td>
</tr>
</tbody>
</table>
MATLAB Vector to Python Mapping

<table>
<thead>
<tr>
<th>MATLAB Input Argument Type — 1-by-N Vector</th>
<th>Resulting Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>array.array('d')</td>
</tr>
<tr>
<td>single</td>
<td>array.array('f')</td>
</tr>
<tr>
<td>int8</td>
<td>array.array('b')</td>
</tr>
<tr>
<td>uint8</td>
<td>array.array('B')</td>
</tr>
<tr>
<td>int16</td>
<td>array.array('h')</td>
</tr>
<tr>
<td>uint16</td>
<td>array.array('H')</td>
</tr>
<tr>
<td>int32</td>
<td>array.array('i')</td>
</tr>
<tr>
<td>uint32</td>
<td>array.array('I')</td>
</tr>
<tr>
<td>int64 (Not supported for Python 2.7 on Windows)</td>
<td>array.array('q')</td>
</tr>
<tr>
<td>uint64 (Not supported for Python 2.7 on Windows)</td>
<td>array.array('Q')</td>
</tr>
<tr>
<td>char array containing values greater than intmax('uint8') (version 2.7 only)</td>
<td>unicode</td>
</tr>
<tr>
<td>char</td>
<td>str</td>
</tr>
<tr>
<td>cell vector</td>
<td>tuple</td>
</tr>
</tbody>
</table>

Unsupported MATLAB Types

The following MATLAB types are not supported in Python.

<table>
<thead>
<tr>
<th>Unsupported MATLAB Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional arrays (numeric, char, or cell)</td>
</tr>
<tr>
<td>Structure arrays</td>
</tr>
<tr>
<td>Complex, scalar integers or arrays</td>
</tr>
<tr>
<td>Sparse arrays</td>
</tr>
<tr>
<td>Logical vectors</td>
</tr>
</tbody>
</table>
Unsupported MATLAB Types

categorical,  
table,  
containers.Map,  
datetime types  
MATLAB objects  
meta.class(py.class)

Related Examples

• “Pass dict Argument to Python Method” on page 1-30

More About

• “Handle Data Returned from Python” on page 1-60
Handle Data Returned from Python

In this section...

“Automatic Python Type to MATLAB Type Mapping” on page 1-60
“Explicit Type Conversions” on page 1-60

Automatic Python Type to MATLAB Type Mapping

The following table shows how MATLAB converts data returned from Python into MATLAB types.

<table>
<thead>
<tr>
<th>Python Return Type, as Displayed in Python</th>
<th>Resulting MATLAB Type — Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical</td>
</tr>
<tr>
<td>int (version 2.7 only). For Python versions 3.x int, you must convert explicitly.</td>
<td>int64</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>complex</td>
<td>Complex double</td>
</tr>
<tr>
<td>All other Python types — type</td>
<td>Python object — py.type</td>
</tr>
</tbody>
</table>

Explicit Type Conversions

MATLAB provides the following functions to convert Python data types to MATLAB types manually.

<table>
<thead>
<tr>
<th>Python Return Type or Protocol, as Displayed in MATLAB</th>
<th>MATLAB Conversion Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>py.str (version 2.7)</td>
<td>char</td>
</tr>
<tr>
<td></td>
<td>uint8</td>
</tr>
<tr>
<td>py.str (version 3.x)</td>
<td>char</td>
</tr>
<tr>
<td>py.unicode</td>
<td>char</td>
</tr>
<tr>
<td>Object with <strong>str</strong> method</td>
<td>char</td>
</tr>
<tr>
<td>py.bytes</td>
<td>uint8</td>
</tr>
</tbody>
</table>
### Python Return Type or Protocol, as Displayed in MATLAB

<table>
<thead>
<tr>
<th>Python Return Type or Protocol, as Displayed in MATLAB</th>
<th>MATLAB Conversion Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>py.int</td>
<td>double or int64</td>
</tr>
<tr>
<td>py.long</td>
<td>double or int64</td>
</tr>
<tr>
<td>py.array.array&lt;sup&gt;a&lt;/sup&gt;</td>
<td>numeric double single int8 uint8 int16 uint16 int32 uint32 int64 uint64</td>
</tr>
<tr>
<td>Sequence protocol; for example, py.list and py.tuple</td>
<td>cell</td>
</tr>
<tr>
<td>Mapping protocol; for example, py.dict</td>
<td>struct</td>
</tr>
</tbody>
</table>

<sup>a</sup> You can convert `py.array.array` of any format to the MATLAB type you want.

### More About
- “Pass Data to Python” on page 1-57
# How MATLAB Represents Python Operators

MATLAB supports the following overloaded operators.

<table>
<thead>
<tr>
<th>Python Operator Symbol</th>
<th>Python Methods</th>
<th>MATLAB Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (binary)</td>
<td><strong>add</strong>, <strong>radd</strong></td>
<td>plus,+</td>
</tr>
<tr>
<td>- (binary)</td>
<td><strong>sub</strong>, <strong>rsub</strong></td>
<td>minus,-</td>
</tr>
<tr>
<td>* (binary)</td>
<td><strong>mul</strong>, <strong>rmul</strong></td>
<td>mtimes,*</td>
</tr>
<tr>
<td>/</td>
<td><strong>truediv</strong>, <strong>rtruediv</strong></td>
<td>mrdivide, /</td>
</tr>
<tr>
<td>==</td>
<td><strong>eq</strong></td>
<td>eq, ==</td>
</tr>
<tr>
<td>&gt;</td>
<td><strong>gt</strong></td>
<td>gt, &gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td><strong>lt</strong></td>
<td>lt, &lt;</td>
</tr>
<tr>
<td>!=</td>
<td><strong>ne</strong></td>
<td>ne, ~=</td>
</tr>
<tr>
<td>&gt;=</td>
<td><strong>ge</strong></td>
<td>ge, &gt;=</td>
</tr>
<tr>
<td>&lt;=</td>
<td><strong>le</strong></td>
<td>le, &lt;=</td>
</tr>
<tr>
<td>- (unary)</td>
<td><strong>neg</strong></td>
<td>uminus, -a</td>
</tr>
<tr>
<td>+ (unary)</td>
<td><strong>pos</strong></td>
<td>uplus, +a</td>
</tr>
</tbody>
</table>

The following Python operators are not supported.

<table>
<thead>
<tr>
<th>Python Operator Symbol</th>
<th>Python Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td><strong>mod</strong>, <strong>rmod</strong></td>
</tr>
<tr>
<td>**</td>
<td><strong>pow</strong>, <strong>rpow</strong></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td><strong>lshift</strong>, <strong>rlshift</strong></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td><strong>rshift</strong>, <strong>rrshift</strong></td>
</tr>
<tr>
<td>&amp;</td>
<td><strong>and</strong>, <strong>rand</strong></td>
</tr>
<tr>
<td>^</td>
<td><strong>xor</strong>, <strong>rxor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>// (binary)</td>
<td><strong>floordiv</strong>, <strong>rfloordiv</strong></td>
</tr>
<tr>
<td>+= (unary)</td>
<td><strong>iadd</strong></td>
</tr>
<tr>
<td>Python Operator Symbol</td>
<td>Python Method</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>-= (unary)</td>
<td><strong>isub</strong></td>
</tr>
<tr>
<td>*= (unary)</td>
<td><strong>imul</strong></td>
</tr>
<tr>
<td>/= (unary)</td>
<td><strong>itruediv</strong></td>
</tr>
<tr>
<td>//= (unary)</td>
<td><strong>ifloordiv</strong></td>
</tr>
<tr>
<td>%= (unary)</td>
<td><strong>imod</strong></td>
</tr>
<tr>
<td>**= (unary)</td>
<td><strong>ipow</strong></td>
</tr>
<tr>
<td>&lt;&lt;= (unary)</td>
<td><strong>ilshift</strong></td>
</tr>
<tr>
<td>&gt;&gt;= (unary)</td>
<td><strong>irshift</strong></td>
</tr>
<tr>
<td>&amp;= (unary)</td>
<td><strong>iand</strong></td>
</tr>
<tr>
<td>^= (unary)</td>
<td><strong>ixor</strong></td>
</tr>
<tr>
<td>!= (unary)</td>
<td><strong>ior</strong></td>
</tr>
<tr>
<td>~(unary)</td>
<td><strong>invert</strong></td>
</tr>
</tbody>
</table>
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 statement:

```matlab
feval(obj, x1, ..., xn)
obj(x1, ..., xn)
```

See Also
`feval`
Python import and MATLAB import Commands

In this section...

“Do Not Type “import pythonmodule”” on page 1-65
“Use MATLAB import to Shorten Class or Function Names” on page 1-65

Do Not Type “import pythonmodule”

MATLAB automatically loads Python when you type:

```matlab
py.command
```

Do not type “import pythonmodule” in MATLAB.

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.

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. Replace the following Python statement:

```python
from x import y
```

with the MATLAB command

```matlab
import x.y
```

where `y` is a class name or function name you want to use.

For example, the Python `textwrap` module formats blocks of text.

```python
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.

```matlab
import py.textwrap.wrap
```
S = wrap('This is a string');

**Note:** Do not call:

```python
import py.*
```

If you do, MATLAB calls the Python function instead of the MATLAB function of the same name.

If you call this command, you must call the MATLAB command:

```matlab
clear import
```

**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.

<table>
<thead>
<tr>
<th>Python list — []</th>
<th>MATLAB py.list</th>
</tr>
</thead>
<tbody>
<tr>
<td>['Robert', 'Mary', 'Joseph']</td>
<td>py.list({'Robert','Mary','Joseph'})</td>
</tr>
<tr>
<td>[[1,2],[3,4]]</td>
<td>py.list({py.list([1,2]),py.list([3,4])})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python tuple — ()</th>
<th>MATLAB py.tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>('Robert', 19, 'Biology')</td>
<td>py.tuple({'Robert',19,'Biology'})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python dict — {}</th>
<th>MATLAB py.dict</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'Robert': 357, 'Joe': 391,</td>
<td>py.dict(pyargs(...</td>
</tr>
<tr>
<td>'Mary': 229}</td>
<td>'Robert',357,'Mary',229,'Joe',391))</td>
</tr>
</tbody>
</table>

See Also

pyargs

Related Examples

- “Use Python list Type in MATLAB” on page 1-20
- “Use Python tuple Type in MATLAB” on page 1-26
- “Use Python dict Type in MATLAB” on page 1-31
Limitations to Python Support

<table>
<thead>
<tr>
<th>Features Not Supported in MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closing the Python interpreter while running MATLAB.</td>
</tr>
<tr>
<td>Saving (serializing) Python objects into a MAT-file.</td>
</tr>
<tr>
<td>Interactive Python help (calling <code>py.help</code> without input arguments).</td>
</tr>
<tr>
<td><code>py.input</code> and <code>py.raw_input</code> (version 2.7).</td>
</tr>
<tr>
<td>Accessing static properties of a Python class.</td>
</tr>
<tr>
<td>MATLAB <code>isa</code> function does not recognize virtual inheritance.</td>
</tr>
<tr>
<td>MATLAB class inheritance from a Python class.</td>
</tr>
<tr>
<td>Overloaded attribute access.</td>
</tr>
<tr>
<td>Nested Python classes.</td>
</tr>
<tr>
<td>Modules that start MATLAB in a separate process, for example, the <code>multiprocessing</code> module.</td>
</tr>
<tr>
<td>Modules that read <code>sys.argv</code>, the command-line arguments passed to a Python script, for example, <code>Tkinter</code>.</td>
</tr>
<tr>
<td>Dynamically generated Python classes, for example, <code>collections.namedtuple</code>.</td>
</tr>
<tr>
<td>Dynamically attaching new object attributes. Instead, use <code>py.setattr</code>.</td>
</tr>
<tr>
<td>Class names or other identifiers starting with an underscore (_) character. Instead, use the Python <code>pygetattr</code> and <code>pysetattr</code> functions.</td>
</tr>
<tr>
<td>Python modules generated by the MATLAB Compiler SDK™ product.</td>
</tr>
</tbody>
</table>

More About

- “Python `import` and MATLAB import Commands” on page 1-65
- “Unsupported MATLAB Types” on page 1-58
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.

<table>
<thead>
<tr>
<th>Indexing Features Not Supported in MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of square brackets, [ ].</td>
</tr>
<tr>
<td>Indexing into a container type that does not inherit from collections.Sequence or collections.Mapping.</td>
</tr>
<tr>
<td>Logical indexing.</td>
</tr>
<tr>
<td>Accessing data in a container with an arbitrary array of indices. An index must be of the form start:step:stop.</td>
</tr>
<tr>
<td>Comma-separated lists.</td>
</tr>
<tr>
<td>numel function does not return number of array elements. Returns 1.</td>
</tr>
</tbody>
</table>

More About

- “Matrix 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.

Use this page to help troubleshoot the failure.

### In this section...

- “Python Not Installed” on page 1-70
- “64-bit/32-bit Versions of Python on Windows Platforms” on page 1-70
- “MATLAB Cannot Find Python” on page 1-71
- “Error in User-Defined Python Module” on page 1-71
- “Python Module Not on Python Search Path” on page 1-71
- “Module Name Conflicts” on page 1-72
- “Python Tries to Execute command in Wrong Module” on page 1-72

### Python Not Installed

Python is not installed on your computer. Download and install Python from [https://www.python.org/downloads](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, download a 64-bit version of Python, identified by the name "Windows x86-64 MSI installer". For more information, see “64-bit/32-bit Versions of Python on Windows Platforms” on page 1-53.

On Linux and Mac systems, if you build Python from source files, 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 or a 64-bit version of Python for a 32-bit version of MATLAB.
MATLAB Cannot Find Python

Python is in a nonstandard location. To provide the path to the Python executable, use the `pyversion` function. For example:

```
pyversion C:\Users\uname\WinPython-64bit-3.3.2.1\python-3.3.2.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:

```
pyversion executable
```

Error in User-Defined Python Module

There is an error 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, 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:

```python
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, MATLAB might load the wrong module.

Python Tries to Execute \textit{command} in Wrong Module

If \textit{command} is in a user-defined module, 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

\texttt{pyversion}

More About

\begin{itemize}
\item “System and Configuration Requirements” on page 1-52
\end{itemize}

External Websites

\begin{itemize}
\item 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.

**More About**

- “Handle Python Exceptions” on page 1-74

**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:

```plaintext
try
  py.list('x','y',1)
catch e
  e.message
  if(isa(e,'matlab.exception.PyException'))
    e.ExceptionObject
  end
end
```

```plaintext
ans =
Python Error: TypeError: list() takes at most 1 argument (3 given)
```

```plaintext
ans =
Python tuple with no properties.

(<type 'exceptions.TypeError'>, TypeError('list() takes at most 1 argument (3 given')
```

If MATLAB displays an error message of the following format, refer to your Python documentation for more information.

```plaintext
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.

In this section...

“Python Error: Python class: message” on page 1-75
“Python Module Errors” on page 1-75
“Errors Converting Python Data” on page 1-76

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](http://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](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 1-70.

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 1-60.

For example, although MATLAB supports multidimensional arrays, you can only pass vectors of data to Python.

```matlab
x = py.len([2 3 4; 4 5 6])
```

Error using `py.len`
Conversion of MATLAB 'double' to Python is only supported for 1-N vectors.

**More About**
- “Data Types”
- “Limitations to Python Support” on page 1-68

**External Websites**
- [www.python.org/doc](http://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.

More About

• “Automatic Python Type to MATLAB Type Mapping” on page 1-60
• “Explicit Type Conversions” on page 1-60
Call Python \texttt{eval} Function

This example shows how to evaluate the expression, $x+y$, in Python. To evaluate an expression, pass a Python \texttt{dict} value for the \texttt{globals} namespace parameter.

Read the help for \texttt{eval}.

\begin{verbatim}
py.help('eval')
\end{verbatim}

Help on built-in function \texttt{eval} in module \texttt{__builtin__}:

\begin{verbatim}
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.
\end{verbatim}

Create a Python \texttt{dict} variable for the $x$ and $y$ values.

\begin{verbatim}
workspace = py.dict(pyargs('x',1,'y',6))
workspace =
    Python dict with no properties.
    {'y': 6.0, 'x': 1.0}
\end{verbatim}

Evaluate the expression.

\begin{verbatim}
res = py.eval('x+y',workspace)
res =
    7
\end{verbatim}

Add two numbers without assigning variables. Pass an empty \texttt{dict} value for the \texttt{globals} parameter.

\begin{verbatim}
res = py.eval('1+6',py.dict)
res =
\end{verbatim}
Call Python eval Function

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))
```
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.

<table>
<thead>
<tr>
<th>Python Signature</th>
<th>MATLAB Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(X)</code></td>
<td><code>py.abs(-99)</code></td>
</tr>
<tr>
<td>Argument X is required.</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Python Signature</th>
<th>MATLAB Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>itertools.izip(*iterables)</code></td>
<td>Aggregate elements from two lists. py.itertools.izip(... py.itertools.izip(py.list({1:10}),py.list({'a','b'})))</td>
</tr>
<tr>
<td>The <code>iterables</code> argument is not required, in which case, the function returns a zero length iterator.</td>
<td>Create zero length iterator. py.itertools.izip;</td>
</tr>
</tbody>
</table>

## 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.

<table>
<thead>
<tr>
<th>Python Signature</th>
<th>MATLAB Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print(*objects,sep='',end=\'\n\',file=sys.stdout)</code></td>
<td><code>py.print('string',pyargs('end','--'))</code></td>
</tr>
<tr>
<td><code>sep, end, and file are keyword arguments.</code></td>
<td></td>
</tr>
</tbody>
</table>

Python defines an arbitrary number of keyword arguments by prepending the name with `**` characters.

<table>
<thead>
<tr>
<th>Python Signature</th>
<th>MATLAB Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dict(**kwarg)</code></td>
<td><code>D = py.dict(pyargs('Joe',100,'Jack',101))</code></td>
</tr>
</tbody>
</table>

**Optional Arguments**

An optional argument is a non-required argument.

<table>
<thead>
<tr>
<th>Python Signature</th>
<th>MATLAB Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>random.randrange(start,stop[,step])</code></td>
<td><code>py.random.randrange(1,100)</code></td>
</tr>
<tr>
<td>Argument <code>step</code> is optional.</td>
<td></td>
</tr>
</tbody>
</table>

Optional arguments can have default values. A default value is indicated by an equal sign `=` with the default value.

<table>
<thead>
<tr>
<th>Python Signature</th>
<th>MATLAB Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print(*objects,sep='',end=\'\n\',file=sys.stdout)</code></td>
<td><code>py.print(2,'2')</code></td>
</tr>
<tr>
<td>The default value for <code>file</code> is <code>sys.stdout</code>.</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

`pyargs`
Related Examples

• “Pass Keyword Arguments” on page 1-34
Read and Write MATLAB MAT-Files in C/C++ and Fortran

• “Custom Applications to Access MAT-Files” on page 2-2
• “MAT-File Library and Include Files” on page 2-5
• “What You Need to Build Custom Applications” on page 2-7
• “Copy External Data into MAT-File Format with Standalone Programs” on page 2-8
• “Create MAT-File in C or C++” on page 2-13
• “Read MAT-File in C/C++” on page 2-14
• “Create MAT-File in Fortran” on page 2-15
• “Read MAT-File in Fortran” on page 2-16
• “Work with mxArrays” on page 2-17
• “Table of MAT-File Source Code Files” on page 2-19
• “Build on Mac and Linux Operating Systems” on page 2-21
• “Build on Windows Operating Systems” on page 2-23
• “Share MAT-File Applications” on page 2-24
Custom Applications to Access MAT-Files

In this section...

“Why Write Custom Applications?” on page 2-2
“MAT-File Interface Library” on page 2-3
“Exchanging Data Files Between Platforms” on page 2-4

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, Load, and Delete 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 15-3). 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 “MEX File Creation API”.

2-2
MAT-File Interface Library

The MAT-File Library 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.

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

MAT-File Routines

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

MAT-File C-Only Routines

| matGetFp                      | 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 Library 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 Library and Include Files

MATLAB provides the include and library files needed 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-Function Folders**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Contents</th>
<th>Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows</td>
<td>Include files</td>
<td><code>matlabroot\extern\include</code></td>
</tr>
<tr>
<td></td>
<td>Libraries</td>
<td><code>matlabroot\bin\win32</code> or <code>matlabroot\bin\win64</code></td>
</tr>
<tr>
<td></td>
<td>Examples</td>
<td><code>matlabroot\extern\examples\eng_mat</code></td>
</tr>
<tr>
<td>Mac</td>
<td>Include files</td>
<td><code>matlabroot/extern/include</code></td>
</tr>
<tr>
<td>Linux</td>
<td>Libraries</td>
<td><code>matlabroot/bin/arch</code></td>
</tr>
<tr>
<td></td>
<td>Examples</td>
<td><code>matlabroot/extern/examples/eng_mat</code></td>
</tr>
</tbody>
</table>

**MAT-Function 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, Mac, and Linux systems. The folder contains:

- The `matrix.h` header file that contains a definition of the `mxArray` structure and function prototypes for matrix access routines.
- The `mat.h` header file that contains function prototypes for `mat` routines.

**MAT-Function 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:

- The `libmat.dll` library of MAT-file routines (C/C++ and Fortran)
• The libmx.dll library of array access and creation routines

**Shared Libraries on Mac and Linux Systems**

The bin/arch folder, where arch is your machine's architecture, contains the shared libraries. For example, on Apple Macintosh 64-bit systems, the folder is bin/mac164:

• The libmat.dylib library of MAT-file routines (C/C++ and Fortran)
• The libmx.dylib library of array access and creation routines

**Example Files**

The 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/C++ and Fortran 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 Library and Include Files” on page 2-5
- “Build Engine Applications with IDE” on page 7-21

External Websites
- Supported and Compatible Compilers
Copy External Data into MAT-File Format with Standalone Programs

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. This is the variable name to use 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, Load, and Delete 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.

<table>
<thead>
<tr>
<th>External Data</th>
<th>Variable to Read External Data</th>
<th>mxArray Variable</th>
<th>MATLAB Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array of type double</td>
<td>extData</td>
<td>pVarNum</td>
<td>inputArray</td>
</tr>
<tr>
<td>String</td>
<td>extString</td>
<td>pVarChar</td>
<td>titleString</td>
</tr>
</tbody>
</table>

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

```c
#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:

```c
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

The MAT-File Library uses pointers of type `mxArray` to reference MATLAB data. The following statements declare `pVarNum` and `pVarChar` as pointers to an array of any size or type:

```c
/*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.

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

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

```c
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”.

```c
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 MX Matrix Library. Since the data contains only real values of type `double`, this example uses the `mxGetPr` function:

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

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

```c
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:

```c
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:

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

Clean Up

To close the file:

```c
matClose(pmat);
```

To free memory:

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

Build the Application

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

```bash
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:

```bash
!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:

```matlab
titleString
whos
```

To see the variables, type whos; MATLAB displays:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputArray</td>
<td>3x3</td>
<td>72</td>
<td>double</td>
</tr>
<tr>
<td>titleString</td>
<td>1x43</td>
<td>86</td>
<td>char</td>
</tr>
</tbody>
</table>

Related Examples

- “Table of MAT-File Source Code Files” on page 2-19
Create MAT-File in C or C++

In this section...

“Create MAT-File in C” on page 2-13
“Create MAT-File in C++” on page 2-13

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-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 command prompt, type:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>GlobalDouble</td>
<td>3x3</td>
<td>72</td>
<td>double array (global)</td>
</tr>
<tr>
<td>LocalDouble</td>
<td>3x3</td>
<td>72</td>
<td>double array</td>
</tr>
<tr>
<td>LocalString</td>
<td>1x43</td>
<td>86</td>
<td>char array</td>
</tr>
</tbody>
</table>

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.

Related Examples

- “Table of MAT-File Source Code Files” on page 2-19
Read MAT-File in C/C++

The matdgnsc 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 2-13 example. To run the application, depending on your platform, either double-click its icon or enter matdgnsc at the system prompt.

matdgnsc 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

Related Examples
• “Create MAT-File in Fortran” on page 2-15
• “Table of MAT-File Source Code Files” on page 2-19
Create MAT-File in Fortran

The `matdemo1.F` example creates the MAT-file, `matdemo.mat`. To see the code, you can 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 command prompt, type:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>3x3</td>
<td>72</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>NumericGlobal</td>
<td>3x3</td>
<td>72</td>
<td>double</td>
<td>global</td>
</tr>
<tr>
<td>String</td>
<td>1x33</td>
<td>66</td>
<td>char</td>
<td></td>
</tr>
</tbody>
</table>

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.

Related Examples

- “Read MAT-File in C/C++” on page 2-14
- “Table of MAT-File Source Code Files” on page 2-19
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
```

Related Examples

- “Table of MAT-File Source Code Files” on page 2-19
Work with mxArrays

In this section...

“Read Structures from a MAT-File” on page 2-17
“Read Cell Arrays from a MAT-File” on page 2-18

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, matdgnsc.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 2-14.
• 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 4-12.
• Some MEX examples use functions, such as mexPrintf, from the “MEX Library API” 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:

```matlab
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
```

To calculate the total of the `billing` field, type:

```
!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 3 cell variables and save:

```matlab
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
```

To display the `mxArray` type for the contents of cell `cellvar`, type:

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

0: string
1: numeric class
2: numeric class

**Related Examples**

- “Table of MAT-File Source Code Files” on page 2-19
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, Mac, and Linux systems.

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

```matlab
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 Library and Include Files” on page 2-5
- “Build on Mac and Linux Operating Systems” on page 2-21
- “Build on Windows Operating Systems” on page 2-23

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matcreat.c</code></td>
<td>C program that demonstrates how to use the library routines to create a MAT-file that you can load into MATLAB.</td>
</tr>
<tr>
<td><code>matcreat.cpp</code></td>
<td>C++ version of the <code>matcreat.c</code> program.</td>
</tr>
<tr>
<td><code>matdgnsc.c</code></td>
<td>C program that demonstrates how to use the library routines to read and diagnose a MAT-file.</td>
</tr>
<tr>
<td><code>matdemo1.F</code></td>
<td>Fortran program that demonstrates how to call the MATLAB MAT-file functions from a Fortran program.</td>
</tr>
<tr>
<td><code>matdemo2.F</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>matimport.c</code></td>
<td>C program based on <code>matcreat.c</code> used in the example for writing standalone applications.</td>
</tr>
<tr>
<td><code>matreadstructarray.c</code></td>
<td>C program based on <code>explore.c</code> to read contents of a structure array.</td>
</tr>
<tr>
<td><code>matreadcellarray.c</code></td>
<td>C program based on <code>explore.c</code> to read contents of a cell array.</td>
</tr>
</tbody>
</table>
### Example Description

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

For examples using the Matrix Library, see:

- “Table of MEX File Source Code Files” on page 5-24.
- The explore.c example described in “Using Data Types” on page 4-12.
Build on Mac and Linux Operating Systems

In this section...

“Setting Run-Time Library Path” on page 2-21
“Building the Application” on page 2-22

Setting Run-Time Library Path

At run time, you must tell the Mac and Linux operating system where the API shared libraries reside by setting an environment variable. The Mac 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.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>envvar</th>
<th>pathspec</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Apple Mac</td>
<td>DYLD_LIBRARY_PATH</td>
<td><code>matlabroot/bin/maci64:matlabroot/sys/os/maci64</code></td>
</tr>
<tr>
<td>64-bit Linux</td>
<td>LD_LIBRARY_PATH</td>
<td><code>matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64</code></td>
</tr>
</tbody>
</table>

Using the C Shell

Set the library path using the command:

```bash
setenv envvar pathspec
```

Replace the terms `envvar` and `pathspec` with the appropriate values from the table. For example, on a Macintosh system use:

```bash
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 Macintosh 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.

**Building the Application**

To compile and link the matcreat.c example, use the mex script with the -client engine option.

```
copyfile(fullfile(matlabroot,'extern','examples','eng_mat','matcreat.c'),'.','f')
```

Use the following command to build it:

```
mex -v -client engine matcreat.c
```

If you need 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
Build on Windows Operating Systems

To compile and link MAT-file programs, use the `mex` script with the `-client engine` option.

```bash
copyfile(fullfile(matlabroot,'extern','examples','eng_mat','matcreat.c'),'.','f')
```

Use the following command to build it:

```bash
mex -v -client engine matcreat.c
```

If you need 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**

- “Build Windows Engine Application” on page 7-11
Share MAT-File Applications

MATLAB requires shared library files for building any MAT-file application. You must also distribute the run-time versions of these files along with any MAT-file application that you deploy to another system. Install the appropriate libraries in the \texttt{matlabroot/bin/arch} folder.

**Library File Names by Operating System**

<table>
<thead>
<tr>
<th>Windows</th>
<th>Linux</th>
<th>Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>libmat.dll</td>
<td>libmat.so</td>
<td>libmat.dylib</td>
</tr>
<tr>
<td>libmx.dll</td>
<td>libmx.so</td>
<td>libmx.dylib</td>
</tr>
</tbody>
</table>

In addition to these libraries, you must have all third-party library files that \texttt{libmat} requires. MATLAB uses these additional libraries to support Unicode\textsuperscript{®} character encoding and data compression in MAT-files. These library files must reside in the same folder as \texttt{libmx}. Determine the libraries using the platform-specific methods described in the following table.

**Library Dependency Commands**

<table>
<thead>
<tr>
<th>Windows</th>
<th>Linux</th>
<th>Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>See the following instructions for Dependency Walker</td>
<td>ldd -d libmat.so</td>
<td>otool -L libmat.dylib</td>
</tr>
</tbody>
</table>

On Windows systems, to find library dependencies, use the third-party product Dependency Walker. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module 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. Download the Dependency Walker utility from the website http://www.dependencywalker.com/. See http://www.mathworks.com/matlabcentral/answers/92362-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 file \texttt{matlabroot/bin/win32/libmat.dll} or \texttt{matlabroot/bin/win64/libmat.dll} into Depends window.
Calling C Shared Library Functions from MATLAB

- “Call Functions in Shared Libraries” on page 3-2
- “Limitations to Shared Library Support” on page 3-8
- “Limitations Using Structures” on page 3-13
- “Module Not Found Error” on page 3-15
- “No Matching Signature Error” on page 3-16
- “MATLAB Terminates Unexpectedly When Calling Function in Shared Library” on page 3-17
- “Pass Arguments to Shared Library Functions” on page 3-18
- “Shared Library shrlibsample” on page 3-23
- “Pass String Arguments” on page 3-24
- “Pass Structures” on page 3-26
- “Pass Enumerated Types” on page 3-32
- “Pass Pointers” on page 3-34
- “Pass Arrays” on page 3-39
- “Iterate Through an Array” on page 3-43
- “Pointer Arguments” on page 3-46
- “Structure Arguments” on page 3-49
- “Explore libstruct Objects” on page 3-51
- “MATLAB Prototype Files” on page 3-52
Call Functions in Shared Libraries

In this section...

“What Is a Shared Library?” on page 3-2
“Load and Unload Library” on page 3-3
“View Library Functions” on page 3-4
“Invoke Library Functions” on page 3-6

What Is a Shared Library?

A shared library is a collection of functions dynamically loaded by an application at run

time. This MATLAB interface supports libraries containing functions programmed in any

language, provided the functions have a C interface. MATLAB supports dynamic linking

on all supported platforms.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Shared Library</th>
<th>File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows</td>
<td>dynamic link library file</td>
<td>.dll</td>
</tr>
<tr>
<td>UNIX® and Linux</td>
<td>shared object file</td>
<td>.so</td>
</tr>
<tr>
<td>Apple Macintosh</td>
<td>dynamic shared library</td>
<td>.dylib</td>
</tr>
</tbody>
</table>

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 the Supported and Compatible Compilers website.

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 the topics:
“Load and Unload Library” on page 3-3
“View Library Functions” on page 3-4
“Invoke Library Functions” on page 3-6

If the library function passes arguments, you need to determine the data type passed to and from the function. For information about data, see:

“Pass Arguments to Shared Library Functions” on page 3-18
“Manually Convert Data Passed to Functions” on page 3-21
“Pointer Arguments” on page 3-46
“Structure Arguments” on page 3-49

When you are finished working with the shared library, it is important to unload the library to free memory.

For more information, see “Limitations to Shared Library Support” on page 3-8

**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. When you no longer need the library, unload it from memory to conserve memory usage.

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`.

For example, load the `libmx` library that defines the MATLAB Matrix Library routines. The following command creates the full path for the library header file, `matrix.h`:
Calling C Shared Library Functions from MATLAB

hfile = fullfile(matlabroot,'extern','include','matrix.h);

To load the library, type:
loadlibrary('libmx',hfile)

Use the unloadlibrary function to unload the library and free up memory.

**Note:** If you call loadlibrary on a library that is already loaded, MATLAB displays a message, but does not reload the library. To determine if a library is loaded, use the libisloaded function.

**View Library Functions**

**View Functions in Command Window**

To display information about library functions in the MATLAB Command Window, use the libfunctions command. For example, to see what functions are available in the libmx library, type:

```matlab
if not(libisloaded('libmx'))
    hfile = [matlabroot '\extern\include\matrix.h'];
    loadlibrary('libmx',hfile)
end
libfunctions libmx
```

MATLAB displays (in part):

Functions in library libmx:

```
mxAddField       mxGetScalar
mxArrayToString  mxGetString_730
mxCalcSingleSubscript_730  mxGetUserBits
mxCalloc         mxIsCell
mxCreateCellArray_730  mxIsChar
mxCreateCellMatrix_730  mxIsClass
```

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 3-18. For example, at the command line enter:

```matlab
list = libfunctions('libmx', '-full')
```

MATLAB displays (in part):

```matlab
list =
'[int32, MATLAB array, cstring] mxAddField(MATLAB array, cstring)'
'[cstring, MATLAB array] mxArrayToString(MATLAB array)'
'[uint64, MATLAB array, uint64Ptr] mxCalcSingleSubscript_730(MATLAB array, uint64, uint64Ptr)'
'lib.pointer mxMalloc(uint64, uint64)'
'[MATLAB array, uint64Ptr] mxCreateCellArray_730(uint64, uint64Ptr)'
'MATLAB array mxCreateCellMatrix_730(uint64, uint64)'
```

**View Functions in Window**

To get information about functions in a library, use the `libfunctionsview` function. MATLAB opens a window to display the following information:

<table>
<thead>
<tr>
<th>Heading</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Type</td>
<td>Types the method returns</td>
</tr>
<tr>
<td>Name</td>
<td>Function name</td>
</tr>
<tr>
<td>Arguments</td>
<td>Valid types for input arguments</td>
</tr>
</tbody>
</table>

To see the functions in the `libmx` library, type:

```matlab
if not(libisloaded('libmx'))
    hfile = [matlabroot '\extern\include\matrix.h'];
    loadlibrary('libmx',hfile)
end
libfunctionsview libmx
```

MATLAB displays the following window:
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.

The following example calls functions from the libmx library. To load the library, type:

if not(libisloaded('libmx'))
    hfile = [matlabroot '\extern\include\matrix.h'];
    loadlibrary('libmx', hfile)
end

To create an array y, type:

y = rand(4,7,2);

To get information about y, type:
calllib('libmx','mxGetNumberOfElements',y)

ans =
  56

MATLAB displays the number of elements in the array.

Type:

calllib('libmx','mxGetClassID',y)

ans =
  mxArray
  mxArray

MATLAB displays the class of the array.

For information on how to define the parameter types, see “Pass Arguments to Shared Library Functions” on page 3-18.
Limitations to Shared Library Support

In this section...

“MATLAB Supports C Library Routines” on page 3-8
“Workarounds for Loading C++ Libraries” on page 3-8
“Limitations Using printf Function” on page 3-9
“Bit Fields” on page 3-9
“Enum Declarations” on page 3-10
“Unions Not Supported” on page 3-10
“Compiler Dependencies” on page 3-11
“Limitations Using Pointers” on page 3-11
“Functions with Variable Number of Input Arguments Not Supported” on page 3-12

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. Many homegrown 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.

Workarounds for Loading C++ Libraries

The shared library interface does not support C++ classes or overloaded functions elements. However, if you have source code for the library, you can apply one of the following strategies to load a C++ library using loadlibrary. After editing the source code, rebuild the library.

Declare Functions as extern “C”

For example, the following function prototype from the file shrlibsample.h shows the syntax to use for each function:

```c
#ifdef __cplusplus
extern "C" {
#endif
```
void addMixedTypes(
    short  x,
    int    y,
    double z
);

/* other prototypes may be here */

#ifdef __cplusplus
}
#endif

The following C++ code is not legal C code for the header file:

extern "C" void addMixedTypes(short x,int y,double z);

**Add Module Definition File in Visual Studio**

While building the DLL from C++ code in Microsoft Visual Studio®, add a Module Definition File (.DEF) in the project. At a minimum, the DEF file must contain the following module-definition statements:

- The first statement in the file must be the LIBRARY statement.
- The EXPORTS statement lists the names and, optionally, the ordinal values of the functions exported by the DLL.

For example, if a DLL exports functions `multDoubleArray` and `addMixedTypes`, `module.def` contains:

```
LIBRARY
EXPORTS
multDoubleArray
addMixedTypes
```

**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**

cahr 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:

```c
struct mystruct
{
    union
    {
        struct {char byte1,byte2;};
        short word;
    };
};
```

with:

```c
struct mystruct
{
    short word;
};
```

where on a little-endian based machine, byte1 is \( \text{mod}(f, 256) \), 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 the Supported and Compatible Compilers website. 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 `double*Ptr*Ptr*Ptr` 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 3-52.

More About

• “Limitations Using Structures” on page 3-13
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.

```latex
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.

```latex
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;
};

More About

• "Limitations to Shared Library Support" on page 3-8
**Module Not Found Error**

This error occurs when the shared library has dependencies which MATLAB cannot find.

On Windows systems, to find library dependencies, use the third-party product Dependency Walker. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module 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. Download the Dependency Walker utility from the website http://www.dependencywalker.com/. See http://www.mathworks.com/matlabcentral/answers/92362-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 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)
```
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 http://www.mathworks.com/matlabcentral/answers/96578-why-does-matlab-crash-when-i-make-a-function-call-on-a-dll-in-matlab-7-6-r2008a.

See Also
loadlibrary

Related Examples
• Why does MATLAB crash when I make a function call on a DLL in MATLAB 7.6 (R2008a)?

More About
• “MATLAB Prototype Files” on page 3-52

External Websites
• Calling Conventions
Pass Arguments to Shared Library Functions

In this section...

- “C and MATLAB Equivalent Types” on page 3-18
- “How MATLAB Displays Function Signatures” on page 3-20
- “NULL Pointer” on page 3-21
- “Manually Convert Data Passed to Functions” on page 3-21

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

<table>
<thead>
<tr>
<th>C Type</th>
<th>Equivalent MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, byte</td>
<td>int8</td>
</tr>
<tr>
<td>unsigned char, byte</td>
<td>uint8</td>
</tr>
<tr>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td>unsigned short</td>
<td>uint16</td>
</tr>
<tr>
<td>int</td>
<td>int32</td>
</tr>
<tr>
<td>long (Windows)</td>
<td>int32, long</td>
</tr>
<tr>
<td>long (Linux)</td>
<td>int64, long</td>
</tr>
<tr>
<td>unsigned int</td>
<td>uint32</td>
</tr>
<tr>
<td>unsigned long (Windows)</td>
<td>uint32, long</td>
</tr>
<tr>
<td>unsigned long (Linux)</td>
<td>uint64, long</td>
</tr>
</tbody>
</table>
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 3-46 for information about when to use a `lib.pointer` object instead.

### MATLAB Extended Types

<table>
<thead>
<tr>
<th>C Pointer Type</th>
<th>Argument Data Type</th>
<th>Equivalent MATLAB Type</th>
<th>Example Function in “Shared Library shrlibsamp” on page 3-23</th>
</tr>
</thead>
<tbody>
<tr>
<td>double *</td>
<td>doublePtr</td>
<td>double</td>
<td>addDoubleRef</td>
</tr>
<tr>
<td>float *</td>
<td>singlePtr</td>
<td>single</td>
<td></td>
</tr>
<tr>
<td>intsize * (integer pointer types)</td>
<td>(u)int(size)Ptr For example, int64 * becomes int64Ptr.</td>
<td>(u)int(size)</td>
<td>multiplyShort</td>
</tr>
<tr>
<td>byte[]</td>
<td>int8Ptr</td>
<td>int8</td>
<td></td>
</tr>
<tr>
<td>char[] (null-terminated string passed by value)</td>
<td>cstring</td>
<td>char array (1xn)</td>
<td>stringToUpper</td>
</tr>
<tr>
<td>char ** (array of pointers to strings)</td>
<td>stringPtrPtr</td>
<td>cell array of strings</td>
<td></td>
</tr>
<tr>
<td>enum</td>
<td>enumPtr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type **</td>
<td>typePtrPtr For example, double ** becomes doublePtrPtr.</td>
<td><code>lib.pointer</code> object</td>
<td>allocateStruct</td>
</tr>
</tbody>
</table>
C Pointer Type | Argument Data Type | Equivalent MATLAB Type | Example Function in “Shared Library shrlibs5ample” on page 3-23
--- | --- | --- | ---
void * | voidPtr |  | deallocateStruct
void ** | voidPtrPtr | lib.pointer object | 
struct (C-style structure) | structure | MATLAB struct | addStructFields
mxArray * | MATLAB array | MATLAB array | 
xmArray ** | MATLAB arrayPtr | lib.pointer object | 

**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 3-21.
- 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 shrlibs5ample” on page 3-23.

**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 shrlibsampl e” on page 3-23

More About
• “Structure Arguments” on page 3-49
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:

  ```matlab
  addpath(fullfile(matlabroot,'extern','examples','shrlib'))
  ```

- Make the folder your current working folder:

  ```matlab
  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

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.

```c
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.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>[cstring, cstring]</td>
<td>stringToUpper</td>
<td>(cstring)</td>
</tr>
</tbody>
</table>

Convert MATLAB Character Array to Uppercase

This example shows how to pass a MATLAB character array str to a C function, stringToUpper.

```matlab
str = 'This was a Mixed Case string';
```

Load the library containing the stringToUpper function.

```matlab
if not(libisloaded('shrlibsample'))
```
Pass String Arguments

```matlab
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
loadlibrary('shrlibsample')
end

Pass `str` to the function.

```matlab
res = calllib('shrlibsample','stringToUpper',str)
```

```matlab
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`.

```matlab
str
```

```matlab
str =
This was a Mixed Case string
```

**Related Examples**

- “Shared Library `shrlibsample`” on page 3-23
- “Iterate Through an Array” on page 3-43
Pass Structures

In this section...

“addStructFields and addStructByRef Functions” on page 3-26
“Add Values of Fields in Structure” on page 3-27
“Preconvert MATLAB Structure Before Adding Values” on page 3-28
“Autoconvert Structure Arguments” on page 3-29
“Pass Pointer to Structure” on page 3-30

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.

```c
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.

```c
EXPORTED_FUNCTION double addStructFields(struct c_struct st) {
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

The MATLAB function signature is:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>addStructFields</td>
<td>(struct c_struct)</td>
</tr>
</tbody>
</table>
**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.

```c
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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>[double, c_structPtr]</td>
<td>addStructByRef</td>
<td>(c_structPtr)</td>
</tr>
</tbody>
</table>

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`.

```matlab
sm.p1 = 476;
sm.p2 = -299;
sm.p3 = 1000;
```

Load the library containing the `addStructFields` function.

```matlab
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.

\[
if \ \text{not} (\text{libisloaded('shrlibsample'))}
\]

\[
\quad \text{addpath(fullfile(matlabroot,'extern','examples','shrlib'))}
\]

\[
\quad \text{loadlibrary('shrlibsample')}
\]

\[
end
\]

Convert the fields, which are of type `double`, to match the \( c\_struct \) structure types, `double`, `short`, and `long`.

\[
sc = \text{libstruct('c\_struct',sm)};
\]

Display the field names and values.

\[
\text{get(sc)}
\]

\[
\quad p1: 476
\quad p2: -299
\quad p3: 1000
\]

Add the field values.
calllib('shrlibsampale','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.

```matlab
if not(libisloaded('shrlibsampale'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsampale')
end
```

Create a structure.

```matlab
S.p1 = 476;
S.p2 = -299;
S.p3 = 1000;
```

Call `addStructByRef`.

```matlab
res = calllib('shrlibsampale','addStructByRef',S)
```

```matlab
res =
  1177
```

MATLAB does not modify the contents of structure `S`, since it is not a pointer.

```matlab
S
```

```matlab
S =
   p1: 476
```
Pass Pointer to Structure

This example shows how calling the `addStructByRef` function with a pointer modifies the fields in the input argument.

```matlab
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end

Create a structure of type `c_struct`.

```matlab
S.p1 = 20;
S.p2 = 99;
S.p3 = 3;
```

Create a pointer `sp` to the structure.

```matlab
sp = libpointer('c_struct',S);
sp.Value
```

```matlab
ans =
    p1: 20
    p2: 99
    p3: 3
```

Pass the pointer to the function.

```matlab
res = calllib('shrlibsample','addStructByRef',sp)
```

```matlab
res =
    122
```

When you pass a pointer, the function modifies the fields in the structure it points to.
sp.Value

ans =

    p1: 5.5000
    p2: 1234
    p3: 12345678

See Also
libpointer | libstruct

Related Examples
• “Shared Library shrlibsample” on page 3-23

More About
• “Strategies for Passing Structures” on page 3-49
• “Limitations Using Structures” on page 3-13
Pass Enumerated Types

In this section...

“readEnum Function” on page 3-32
“Display Enumeration Values” on page 3-32

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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstring</td>
<td>readEnum</td>
<td>(Enum1)</td>
</tr>
</tbody>
</table>

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

Related Examples
- “Shared Library shrlibsample” on page 3-23
Pass Pointers

**multDoubleRef Function**

The `multDoubleRef` function in the `shrlibsample` library multiplies the input by 5.

```c
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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>[lib.pointer, doublePtr]</td>
<td>multDoubleRef</td>
<td>(doublePtr)</td>
</tr>
</tbody>
</table>

**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.

```matlab
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```
Pass Pointers

Construct a pointer, Xptr, to the input argument, X.

```matlab
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('shrlibsamp;le','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.

```matlab
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.

```matlab
X = 1:10;
xp = libpointer('doublePtr',X);
xp.Value
```

```
ans =
     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`.

```matlab
xp2 = xp + 4;
xp2.Value
```

```
ans =
     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.

```c
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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_structPtrPtr</td>
<td>allocateStruct</td>
<td>(c_structPtrPtr)</td>
</tr>
<tr>
<td>voidPtr</td>
<td>deallocateStruct</td>
<td>(voidPtr)</td>
</tr>
</tbody>
</table>

**Pass Multilevel Pointer**

This example shows how to pass a multilevel pointer to a C function.

Load the library containing `allocateStruct` and `deallocateStruct`.

```matlab
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Create a `c_structPtr` pointer.

```matlab
sp = libpointer('c_structPtr');
```

Call `allocateStruct` to allocate memory for the structure.

```matlab
res = calllib('shrlibsample','allocateStruct',sp)
```

```matlab
res =
    p1: 12.4000
    p2: 222
    p3: 333333
```

Free the memory created by the `allocateStruct` function.

```matlab
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:
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

In this section...

"print2darray Function" on page 3-39
"Convert MATLAB Array to C-Style Dimensions" on page 3-39
"multDoubleArray Function" on page 3-40
"Preserve 3-D MATLAB Array" on page 3-41

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 =

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.

```c
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:
### Pass Arrays

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>doublePtr</td>
<td>multDoubleArray</td>
<td>(doublePtr, int32)</td>
</tr>
</tbody>
</table>

#### 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.

```matlab
if not(libisloaded('shrlibsampl'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsampl')
end
```

Create a 2-by-5-by-2 input array and display its dimensions.

```matlab
vin = reshape(1:20,2,5,2);
vs = size(vin)
```

```matlab
vs =
    2     5     2
```

Call `multDoubleArray` to multiply each element. Display the dimensions of the output.

```matlab
vout = calllib('shrlibsampl','multDoubleArray',vin,20);
size(vout)
```

```matlab
ans =
    2   10
```

Restore the original shape.

```matlab
vout = reshape(vout,vs);
size(vout)
```
ans =
   2   5   2
Iterate Through an Array

In this section...

“Create Cell Array from lib.pointer Object” on page 3-43
“Perform Pointer Arithmetic on Structure Array” on page 3-44

Create Cell Array from lib.pointer Object

This example shows how to create a MATLAB® cell array of strings, `mlStringArray`, from the output of the `getListOfStrings` function.

Load the `shrlibsample` library.

```matlab
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Call the `getListOfStrings` function to create an array of strings. The function returns a pointer to the array.

```matlab
ptr = calllib('shrlibsample','getListOfStrings');
class(ptr)
```

```matlab
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.

```matlab
ptrindex = ptr;
index = 1;
```

Create the cell array of strings `mlStringArray`. Copy the output of `getListOfStrings` to the cell array.

```matlab
% 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 =
"String 1"    "String Two"    ""    "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.

```matlab
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
```

Create the MATLAB structure.

```matlab
s = struct('p1',{1,2,3},'p2',{1.1,2.2,3.3},'p3',{0});
```

Create a pointer to the structure.

```matlab
sptr = libpointer('c_struct',s);
```

Read the values of the first element.

```matlab
v1 = sptr.Value
```

v1 =
Iterate Through an Array

p1: 1
p2: 1
p3: 0

Read the values of the next element by incrementing the pointer.

sptr = sptr + 1;
v2 = sptr.Value

v2 =

p1: 2
p2: 2
p3: 0
Pointer Arguments

### In this section...

- "Pointer Arguments in C Functions" on page 3-46
- "Put String into Void Pointer" on page 3-46
- "Memory Allocation for External Library" on page 3-47

## 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 `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:

```matlab
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:

```matlab
char(vp.Value)
vp.DataType
```
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)`
- `IsAquisitionDone(void)`

where `buffer` is declared as follows:

```c
short buffer[99]
```

First, create a `lib.pointer` to an array of 99 points:

```c
BufferSize = 99;
pBuffer = libpointer('int16Ptr', zeros(BufferSize, 1));
```

Then, begin acquiring data and wait in a loop until it is done:

```c
calllib('myLib', 'AcquireData', BufferSize, pBuffer)
while (~calllib('myLib', 'IsAquisitionDone'))
    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 3-27
• “Preconvert MATLAB Structure Before Adding Values” on page 3-28

More About
• “Limitations Using Structures” on page 3-13
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.

```matlab
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.

```matlab
sc = libstruct('c_struct')
```

```matlab
sc =
    lib.c_struct
```

Set structure field values.

```matlab
set(sc,'p1',100,'p2',150,'p3',200);
```

Display field values.

```matlab
get(sc)
```

```matlab
    p1: 100
    p2: 150
    p3: 200
```

Modify values using MATLAB field structure syntax.

```matlab
sc.p1 = 23;
get(sc)
```

```matlab
    p1: 23
    p2: 150
    p3: 200
```
MATLAB Prototype Files

In this section...

“When to Use Prototype Files” on page 3-52
“How to Create Prototype Files” on page 3-52
“How to Specify Thunk Files” on page 3-53
“Deploy Applications That Use loadlibrary” on page 3-53
“loadlibrary in Parallel Computing Environment” on page 3-53
“Change Function Signature” on page 3-53
“Rename Library Function” on page 3-53
“Load Subset of Functions in Library” on page 3-53
“Call Function with Variable Number of Arguments” on page 3-54

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

• “Introducing MEX Files” on page 4-3
• “Using MEX Files” on page 4-4
• “MEX File Placement” on page 4-5
• “Use Help Files with MEX Files” on page 4-6
• “MATLAB Data” on page 4-7
• “Testing for Most-Derived Class” on page 4-15
• “Build MEX File” on page 4-17
• “Linking Multiple Files” on page 4-18
• “What You Need to Build MEX Files” on page 4-19
• “Change Default Compiler” on page 4-20
• “Custom Build with MEX Script Options” on page 4-23
• “Compiling MEX Files with the Microsoft Visual C++ IDE” on page 4-24
• “Call LAPACK and BLAS Functions” on page 4-26
• “Running MEX Files with .DLL File Extensions on Windows 32-Bit Platforms” on page 4-35
• “Upgrade MEX-Files to Use 64-Bit API” on page 4-36
• “Upgrade MEX Files to Use Graphics Objects” on page 4-47
• “Platform Compatibility” on page 4-52
• “Invalid MEX File Error” on page 4-53
• “Run MEX File You Receive from Someone Else” on page 4-54
• “MEX File Dependent Libraries” on page 4-55
• “Document Build Information in the MEX File” on page 4-56
• “Version Compatibility” on page 4-58
• “Getting Help When MEX Fails” on page 4-59
• “Understanding MEX File Problems” on page 4-61
• “Compiler- and Platform-Specific Issues” on page 4-66
• “Memory Management Issues” on page 4-67
• “Compiler Errors in Fortran MEX Files” on page 4-73
Introducing MEX Files

You can call your own C, C++, or Fortran subroutines from the MATLAB command line as if they were built-in functions. These programs, called binary MEX files, are dynamically linked subroutines that the MATLAB interpreter loads and executes. The MEX file contains only one function or subroutine, and its name is the MEX file name. To call a MEX file, use the name of the file, without the file extension.

For information about using a MEX file that someone else created, see “Call MEX File Functions”.

The term mex stands for “MATLAB executable” and has different meanings, as shown in the following table.

<table>
<thead>
<tr>
<th>MEX Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>source MEX file</td>
<td>C, C++, or Fortran source code file.</td>
</tr>
<tr>
<td>binary MEX file</td>
<td>Dynamically linked subroutine executed in the MATLAB environment.</td>
</tr>
<tr>
<td>MEX function library</td>
<td>MATLAB C/C++ and Fortran API Reference library to perform operations in the MATLAB environment.</td>
</tr>
<tr>
<td>mex build script</td>
<td>MATLAB function to create a binary file from a source file.</td>
</tr>
</tbody>
</table>

Related Examples

- “Create C Source MEX File” on page 5-18

More About

- “Creating C++ MEX Files” on page 5-13
- “What You Need to Build MEX Files” on page 4-19
Using MEX Files

Binary MEX files are subroutines produced from C/C++ or Fortran source code. They behave just like MATLAB scripts and built-in functions. While scripts have a platform-independent extension `.m`, MATLAB identifies MEX files by platform-specific extensions. The following table lists the platform-specific extensions for MEX files.

### MEX-File Platform-Dependent Extension

<table>
<thead>
<tr>
<th>Platform</th>
<th>Binary MEX-File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (64-bit)</td>
<td>mexa64</td>
</tr>
<tr>
<td>Apple Mac (64-bit)</td>
<td>mexmaci64</td>
</tr>
<tr>
<td>Microsoft Windows (32-bit)</td>
<td>mexw32</td>
</tr>
<tr>
<td>Windows (64-bit)</td>
<td>mexw64</td>
</tr>
</tbody>
</table>

You cannot use a binary MEX file on a platform if you compiled it on a different platform. Recompile the source code on the platform for which you want to use the MEX file. For information about using MEX S-functions, see your Simulink® documentation.
**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 find that 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 Files

You can document the behavior of your MEX files 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`.

```matlab
% 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 4-56
- “Add Help for Your Program”
MATLAB Data

In this section...

“The MATLAB Array” on page 4-7
“Lifecycle of mxArray” on page 4-7
“Data Storage” on page 4-8
“MATLAB Types” on page 4-10
“Sparse Matrices” on page 4-11
“Using Data Types” on page 4-12

The MATLAB Array

The MATLAB language works with a single object type: the MATLAB array. All MATLAB variables (including scalars, vectors, matrices, strings, cell arrays, structures, and objects) are stored as MATLAB arrays. In C/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 object, the number of fields and field names

To access the mxArray structure, use the API functions in the Matrix Library. These functions allow you to create, read, and query information about the MATLAB data in your MEX files. Matrix Library functions use the mwSize type 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 “Components of MEX File” on page 5-3 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, which is listed in the create function’s MX Matrix Library documentation.

**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. Additionally, *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 *mxCalloc* 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*. Refer to the MX Matrix Library function documentation for information about which function to use.

**Data Storage**

MATLAB stores data in a column-major (columnwise) 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']

ans =
    house
    floor
    porch

its dimensions are:

size(a)

ans =
    3     5

and its data is stored as:

```
   h     f     p     o     l     o
   u     u     o     r     s     o
   c     r     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:

```
   Q     R
   S     T
   U     V
   -X
```

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
```
The `mxCalcSingleSubscript` function creates the offset from the first element of an array to the desired element, using N-dimensional subscripting.

**MATLAB 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 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.

**Other Numeric Matrices**

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

**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 Strings**

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

**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 \texttt{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 of the data is the same as matrices.

Empty Arrays

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

Sparse Matrices

Sparse matrices have a different storage convention from full matrices in MATLAB. The parameters \( pr \) and \( pi \) are still arrays of double-precision numbers, but these arrays contain only nonzero data elements. There are three additional parameters: \texttt{nzmax}, \texttt{ir}, and \texttt{jc}.

- \texttt{nzmax} is an integer that contains the length of \( ir \), \( pr \), and \( pi \), if it exists. It is the maximum number of nonzero elements in the sparse matrix.
- \texttt{ir} points to an integer array of length \texttt{nzmax} containing the row indices of the corresponding elements in \( pr \) and \( pi \).
- \texttt{jc} points to an integer array of length \( n+1 \), where \( n \) is the number of columns in the sparse matrix. The \texttt{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 \( ir \), \( pr \), and \( pi \) 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,
mxCreateStructArray). 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 mxArray. The following statement declares an
mxArray named myData:

```
mxArray *myData;
```

To define the values of myData, use one of the mxCreate* functions. Some useful
array creation routines are mxCreateNumericArray, mxCreateCellArray, and
mxCreateCharArray. For example, the following statement allocates an m-by-1 floating-
point mxArray initialized to 0:

```
myData = mxCreateDoubleMatrix(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 4-8.) Use the MATLAB mxGet*
array access routines to read data from an mxArray.

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 mxGetData, mxGetPr, 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 string `prhs[0]` into a C-style string `buf`:

```c
char *buf;
int buflen;
int status;
buflen = mxGetN(prhs[0])*sizeof(mxChar)+1;
buf = mxMalloc(buflen);
status = mxGetString(prhs[0], buf, buflen);
```

**The 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.

```bash
copyfile(fullfile(matlabroot,'extern','examples','mex','explore.c'),'.','f')
```

Use the `mex` function to build the MEX file.

```bash
mex -largeArrayDims explore.c
```

Type:

```matlab
x = 2;
expose(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)
```
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:

```matlab
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:

```matlab
outArray = myMexFcn(a,b)
```

Define a cell array floatTypes that contains the strings double and single:

```matlab
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:

```matlab
% 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:

```matlab
if isa(a,'integer') && ~isobject(a)
    % a is a built-in integer type
    ...
end
```
Build 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 on your path.

```matlab
copyfile(fullfile(matlabroot,'extern','examples','refbook','timestwo.c'),'.','f')
```

Use the `mex` function to build the MEX file.

```matlab
mex  timestwo.c
```

Building with 'Microsoft Visual C++ 2010 (C)'.
MEX completed successfully.

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` like a MATLAB function.

```matlab
  timestwo(4)
```

ans =
   8

**See Also**

`mex` | `mexext`

**More About**

- “What You Need to Build MEX Files” on page 4-19
- “Upgrade MEX-Files to Use 64-Bit API” on page 4-36
**Linking Multiple Files**

You can combine multiple source files, object files, and file libraries to build a binary MEX file. List the additional files, with their file extensions, separated by spaces. The name of the MEX file is the name of the first file in the list.

The following command combines multiple files of different types into a binary MEX file called `circle.ext`, where `ext` is the extension corresponding to the current platform:

```
mex circle.c square.obj rectangle.c shapes.lib
```

For a Fortran files, type:

```
mex circle.F square.o rectangle.F shapes.o
```

You can use a software development tool like MAKE to manage MEX file projects involving multiple source files. Create a MAKFILE that contains a rule for producing object files from each of your source files. Then invoke the `mex` build script to combine your object files into a binary MEX file. This method ensures that your source files are recompiled only when necessary.
What You Need to Build MEX Files

To create a MEX file:

• Install a MATLAB-supported compiler.
• Assemble your functions and the MATLAB API functions into one or more C/C++ or Fortran source files.
• Write a gateway function in one of your source files.
• Use the MATLAB \textit{mex} function, called a build script, to build a binary MEX file.
• Use your binary MEX file like any MATLAB function.

If you have multiple compilers installed on your system, see “Change Default Compiler” on page 4-20.

See Also
\texttt{mex}

Related Examples
• “Build MEX File” on page 4-17

More About
• “C/C++ Matrix Library API”
• “Fortran Matrix Library API”
• “Troubleshoot MEX Files”

External Websites
• Supported and Compatible Compilers
Change Default Compiler

In this section...

“Windows Systems” on page 4-20
“Mac and Linux Systems” on page 4-21
“Do Not Use mex -f optionsfile Syntax” on page 4-21

Windows Systems

If you have multiple MATLAB-supported compilers for a language installed on your Windows system, MATLAB selects one as the default compiler. You can change the default using the `mex -setup language` command. If you have multiple compilers, MATLAB displays a message with links to set up a different compiler.

If you call `mex -setup` without the `language` argument, MATLAB displays links to the other supported languages. Select a link to change the default for building MEX files in that language. MATLAB maintains separate default compiler options for C language and C++ language files.

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 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

**Mac and Linux Systems**

MATLAB supports only one compiler for each language on Linux and Mac platforms. If you have multiple compilers installed, the default compiler might not be the MATLAB-supported compiler. You can either change the system default compiler for all applications, or select the MATLAB-supported compiler each time you run the `mex` command.

**Change System Default Compiler**

To determine the default gcc compiler for your system, in MATLAB, type:

```matlab
!which gcc
```

If this compiler is not a MATLAB-supported compiler, you can change the default. When you change the compiler, it becomes the default compiler for all applications on your system. To change the system default compiler:

**Select MATLAB-Supported Compiler When Running mex**

To change the compiler in the `mex` command, set the `varname` variable. `varname` for the gcc compiler is `GCC`, in uppercase letters. For example, if the currently supported gcc compiler is version 4.7, and it is installed in the `/usr/bin/gcc-4.7` folder on your system, to build `yprime.c`, type:

```matlab
copyfile(fullfile(matlabroot,'extern','examples','refbook','timestwo.c'),'.','f')
mex -v GCC='/usr/bin/gcc-4.7' timestwo.c
```

**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 work flows described in this topic for specifying a compiler.

**See Also**

mex

**Related Examples**

- “Choose a C++ Compiler” on page 5-28
External Websites

• Supported and Compatible Compilers
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`
Compiling MEX Files with the Microsoft Visual C++ IDE

Note: This topic provides information on how to compile source MEX files in the Microsoft Visual C++ IDE. It assumes that you know how to use the IDE. If you need more information, refer to the corresponding Microsoft documentation.

To build MEX files with the Microsoft Visual C++ integrated development environment:

1. Create a project and insert your MEX source files.
2. Create a .def file to export the MEX entry point. On the Project menu, click Add New Item and select Module-Definition File (.def). For example:

   LIBRARY MYFILE
   EXPORTS mexFunction        <-- for a C MEX-file
   or
   EXPORTS _MEXFUNCTION       <-- for a Fortran MEX-file

3. On the Project menu, click Properties for the project to open the property pages.
4. Under C/C++ General properties, add the MATLAB include folder, matlab\extern\include, as an additional include folder.
5. Under C/C++ Preprocessor properties, add MATLAB_MEX_FILE as a preprocessor definition.
6. Under Linker General properties, change the output file extension. If you are building for a 32-bit platform, use .mexw32. If you are building for a 64-bit platform, use .mexw64.
7. Locate the .lib files for the compiler you are using under matlabroot\extern\lib\win32\microsoft or matlabroot\extern\lib\win64\microsoft. Under Linker Input properties, add libmx.lib, libmex.lib, and libmat.lib as additional dependencies.
8. Under Linker Input properties, add the module definition (.def) file you created.
9. Under Linker Debugging properties, if you intend to debug the MEX file using the IDE, specify that the build generates debugging information. For more information about debugging, see “Debugging on Microsoft Windows Platforms” on page 5-52.

If you are using a compiler other than the Microsoft Visual C++ compiler, the process for building MEX files is similar. In step 4, locate the .lib files for the compiler you are
using in a folder of `matlabroot\extern\lib\win32` or `matlabroot\extern\lib\win64`. 
Call LAPACK and BLAS Functions

In this section...

“What You Need to Know” on page 4-26
“Creating a MEX File Using LAPACK and BLAS Functions” on page 4-26
“Preserving Input Values from Modification” on page 4-28
“Passing Arguments to Fortran Functions from C/C++ Programs” on page 4-29
“Passing Arguments to Fortran Functions from Fortran Programs” on page 4-30
“Handling Complex Numbers in LAPACK and BLAS Functions” on page 4-31
“Modifying the Function Name on UNIX Systems” on page 4-34

What You Need to Know

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 work with complex numbers, use the conversion routines in the fort.c and fort.h files in matlabroot/extern/examples/refbook. To help you get started, there are source code examples in matlabroot/extern/examples/refbook.

Creating a MEX File Using LAPACK and BLAS Functions

To call LAPACK or BLAS functions:

1. Create a source MEX file containing the mexFunction gateway routine.
2. Select a supported compiler for your platform.
3. Build a binary MEX file using the mex command with one or more of the following options:
   - Link your source file to one or both of the libraries, mwlapack and mwblas.
   - Use the -largeArrayDims option; the mwlapack and mwblas libraries only support 64-bit integers for matrix dimensions.
If your function uses complex numbers, build your source file with `fort.c` and include the `fort.h` header file.

The following topics show how to use the `mex` command using the example `matrixMultiply.c`. To work with this file, copy it to a local folder. For example:

```matlab
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:

```matlab
fileattrib('matrixMultiply.c','+w')
```

### Building on Windows Platforms

There are compiler-specific versions of the libraries on the Windows platform. To link to a specific library, look at the `matlabroot/extern/lib/` folder and choose the path for your architecture and compiler.

For example, to link to the libraries for a Microsoft C/C++ compiler, create the following build command variables, `lapacklib` and `blaslib`. These variables identify the full path and file name of each library.

```matlab
lapacklib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft','...','libmwlapack.lib');
blaslib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft','...','libmwblas.lib');
```

When you use a variable to identify the library, use the function syntax of the `mex` command. (For more information, see “Command vs. Function Syntax”.) To build `matrixMultiply.c`, which uses functions from the BLAS library, type:

```matlab
mex('-v', '-largeArrayDims', 'matrixMultiply.c', blaslib)
```

To build a MEX file with functions that use complex numbers, see “Handling Complex Numbers in LAPACK and BLAS Functions” on page 4-31.

### Building on UNIX Platforms

To build the MEX file `matrixMultiply.c`, which uses functions from the BLAS library, type:

```bash
mex -v -largeArrayDims matrixMultiply.c -lmwblas
```
To build a MEX file with functions that use complex numbers, see “Handling Complex Numbers in LAPACK and BLAS Functions” on page 4-31.

Testing the matrixMultiply MEX File

To run the matrixMultiply MEX file, type:

\[
A = \begin{bmatrix} 1 & 3 & 5; 2 & 4 & 7 \end{bmatrix};  \\
B = \begin{bmatrix} -5 & 8 & 11; 3 & 9 & 21; 4 & 0 & 8 \end{bmatrix};  \\
X = \text{matrixMultiply}(A,B)
\]

\[
X = \\
\begin{bmatrix} 24 & 35 & 114; 30 & 52 & 162 \end{bmatrix}
\]

Preserving 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 5-4.

Example — matrixDivide.c

The following 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 the file in the MATLAB Editor. To create the MEX file, copy the source file to a writable folder.

\[
\text{copyfile}()\text{fullfile(matlabroot,'extern','examples','refbook','matrixDivide.c'),'.}')\]

To build the file on Windows, type:

\[
\text{lapacklib} = \text{fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft',...}
\text{'libmwlapack.lib'});  \\
\text{mex('-v', '-largeArrayDims', 'matrixDivide.c', lapacklib)}
\]

To build the file on UNIX type:

\[
\text{mex} -v -largeArrayDims matrixDivide.c -lmwlapack
\]
To test, type:

```matlab
A = [1 2; 3 4];
B = [5; 6];
X = matrixDivide(A,B)
```

```
X =
-4.0000
4.5000
```

**Passing 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 expect the arguments to be passed by reference, while arguments to C/C++ functions are passed 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 do this, precede the argument with an ampersand (&), unless that argument is already a reference. For example, when you create a matrix using the `mxGetPr` 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.

```c
/* 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);
```

**Example — matrixMultiply.c**

The `matrixMultiply.c` example calls `dgemm`, passing all arguments by reference. To see the source code, open the file in the MATLAB Editor. To build and run this example, see “Creating a MEX File Using LAPACK and BLAS Functions” on page 4-26.
Passing 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 \texttt{dgemm}. To run the example, copy the code into the editor and name the file \texttt{calldgemm.F}.

```fortran
#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 mxcopyptrtoereal8(A, ar, numel)
  call mxcopyptrtoereal8(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 \texttt{dgemm} function.
blaslib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft', ...
    'libmwbblas.lib');

To build the file, type:

mex('-v','-largeArrayDims','calldgemm.F',blaslib)

Handling Complex Numbers in LAPACK and BLAS Functions

MATLAB stores complex numbers differently than Fortran. MATLAB stores the real and imaginary parts of a complex number in separate, equal length vectors, pr and pi. Fortran stores the same complex number in one location with the real and imaginary parts interleaved.

As a result, complex variables exchanged between MATLAB and a Fortran function are incompatible. Use the conversion routines, mat2fort and fort2mat, that change the storage format of complex numbers to address this incompatibility.

- mat2fort — Convert MATLAB complex matrix to Fortran complex storage.
- fort2mat — Convert Fortran complex storage to MATLAB real and imaginary parts.

The fort.c and fort.h files provide routines for conversion between MATLAB and Fortran complex data structures. These files define the mat2fort and fort2mat routines.

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.

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 “Preserving Input Values from Modification” on page 4-28.
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 \texttt{zout} in the example.

2. To make the variable compatible with MATLAB, use the \texttt{fort2mat} function.

Example — Passing Complex Variables

This example shows how to call a function, passing complex \texttt{prhs[0]} as input and receiving complex \texttt{plhs[0]} as output. Temporary variables \texttt{zin} and \texttt{zout} contain the input and output values in Fortran format. To see the example, open the file in the MATLAB Editor. To create the MEX file, copy the source file to a writable folder.

\begin{verbatim}
Example of passing complex variables.

To build the file on a Windows platform, type:

\begin{verbatim}
lapacklib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft',
                'libmwlapack.lib');
fortfile = fullfile(matlabroot,'extern','examples','refbook','fort.c');
fortheaderdir = fullfile(matlabroot,'extern','examples','refbook');
mex('-v', '-largeArrayDims', ['-I' fortheaderdir], ...
     'matrixDivideComplex.c', fortfile, lapacklib)
\end{verbatim}

To build on a UNIX platform, type:

\begin{verbatim}
fortfile = fullfile(matlabroot,'extern','examples','refbook','fort.c');
fortheaderdir = fullfile(matlabroot,'extern','examples','refbook');
mex('-v', '-largeArrayDims', ['-I' fortheaderdir], ...
     'matrixDivideComplex.c', fortfile, '-lmwlapack')
\end{verbatim}

To test:

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)
\end{verbatim}
\[ X = \\
\begin{align*}
-4.4000 & + 0.8000i \\
4.8000 & - 0.6000i
\end{align*}
\]

Example — Handling Fortran Complex Return Type

Some level 1 BLAS functions (for example, \texttt{zdotu} and \texttt{zdotc}) return a \textbf{double complex} type, which the C language does not support. The following C MEX file, \texttt{dotProductComplex.c}, shows how to handle the Fortran complex return type for function \texttt{zdotu}. To see the example, open the file 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 platform, pass the return value as the first input argument. MATLAB provides a macro, \texttt{FORTRAN_COMPLEX_FUNCTIONS_RETURN_VOID}, to handle these differences.

The \texttt{dotProductComplex} example computes the dot product \( X \) of each element of two complex vectors \( A \) and \( B \). The calling syntax is:

\[ X = \text{dotProductComplex}(A,B) \]

where \( A \) and \( B \) are complex vectors of the same size and \( X \) is a complex scalar.

For example, to build the MEX file on a Windows platform, type:

\begin{verbatim}
blaslib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft',...'libmwblas.lib');
fortfile = fullfile(matlabroot,'extern','examples','refbook','fort.c');
fortheaderdir = fullfile(matlabroot,'extern','examples','refbook');
mex('-v', '-largeArrayDims', ['-I' fortheaderdir], 'dotProductComplex.c', fortfile, blaslib)
\end{verbatim}

To test, type:

\begin{verbatim}
a1 = [1+2i; 2+3i];
b1 = [-1+2i; -1+3i];
X = dotProductComplex(a1,b1)
\end{verbatim}

\[ X = \\
\begin{align*}
-16.0000 & + 3.0000i
\end{align*}
\]

Example — Symmetric Indefinite Factorization Using LAPACK

The example \texttt{utdu_slv.c} calls LAPACK functions \texttt{zhesvx} and \texttt{dsysvx}. To see the example, open the file 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 the file on Windows, type:

```matlab
lapacklib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft',... 'libmwlapack.lib');
fortheaderdir = fullfile(matlabroot,'extern','examples','refbook');
mex('-v','-largeArrayDims', ['-I' fortheaderdir],'utdu_slv.c',fortfile,lapacklib)
```

To build on a UNIX platform, type:

```matlab
mex -v -largeArrayDims utdu_slv.c -lmwlapack
```

**Modifying the 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:

```matlab
dgemm_(arg1, arg2, ..., argn);
```

Or add these lines to your source code:

```matlab
#if !defined(_WIN32)
#define dgemm dgemm_
#endif
```
Running MEX Files with .DLL File Extensions on Windows 32-Bit Platforms

A MEX file is a shared library dynamically loaded at runtime. Shared libraries are sometimes called .dll files, for dynamically linked library. MEX files have a platform-dependent extension, which the `mex` function automatically assigns.

On 32-bit Windows platforms, the MEX file extension is `.mexw32`. MATLAB also supports `.dll` as a secondary MEX file extension. However, future versions of MATLAB will not support this extension.

To convert a MEX file to the `.mexw32` file extension, rebuild the source file.

More About
- “Call MEX File Functions”
- “Build MEX File” on page 4-17
Upgrade MEX-Files to Use 64-Bit API

In this section...

“MATLAB Support for 64-Bit Indexing” on page 4-36
“MEX Uses 32-Bit API by Default” on page 4-36
“What If I Do Not Upgrade?” on page 4-38
“How to Upgrade MEX-Files to Use the 64-Bit API” on page 4-38

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 \texttt{int} in C/C++ and \texttt{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 MX Matrix Library support 64-bit indexing:

- New types, \texttt{mwSize} and \texttt{mwIndex}, enabling large-sized data.
- Updated MX Matrix Library functions use \texttt{mwSize} and \texttt{mwIndex} types for inputs and outputs. These functions are called the 64-bit API or the large-array-handling API.
- New \texttt{-largeArrayDims} flag for \texttt{mex} build command enabling use of the 64-bit 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 \texttt{-compatibleArrayDims} flag.

Note: Only variables representing array size or index value require the \texttt{mwSize} or \texttt{mwIndex} types. The C-language \texttt{int} data type is valid for variables representing, for example, the number of fields or arrays.

MEX Uses 32-Bit API by Default

The \texttt{mex} command uses the \texttt{-compatibleArrayDims} flag (32-bit API) by default. In a future version of MATLAB, the \texttt{mex} command will change to use the large-array-
handling API. Then, the \texttt{-largeArrayDims} option will be the default. This topic describes how to upgrade your MEX-files now in preparation for that transition.

**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, unrelated incompatibilities that prevent execution of an existing MEX-file can occur. If your MEX-file does not execute properly, review the MEX Compatibility Considerations topics in the Release Notes for this release. To find MEX topics, check the External Interfaces section of the Compatibility Summary for MATLAB release notes table for each relevant version.

**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 when the default changes. To upgrade, review your source code, make appropriate changes, and rebuild using the \texttt{mex} command.

Previous versions of the External Interfaces Release Notes provide instructions for updating your MEX-files. 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.

<table>
<thead>
<tr>
<th>State of Your Source Code</th>
<th>Next Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>I do not plan to update my code.</td>
<td>You have chosen to opt out and you must build using the \texttt{-compatibleArrayDims} flag.</td>
</tr>
<tr>
<td>I want to update my code. Where do I start?</td>
<td>See “How to Upgrade MEX-Files to Use the 64-Bit API” on page 4-38.</td>
</tr>
<tr>
<td>I use MEX-files, but do not have access to the source code.</td>
<td>Ask the owner of the source code to follow the steps in “How to Upgrade MEX-Files to Use the 64-Bit API” on page 4-38.</td>
</tr>
<tr>
<td>I use third-party libraries.</td>
<td>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 \texttt{-compatibleArrayDims} flag.</td>
</tr>
<tr>
<td></td>
<td>If the libraries support 64-bit indexing, review your source code, following the steps</td>
</tr>
<tr>
<td>State of Your Source Code</td>
<td>Next Action</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>I updated my code in a previous release.</td>
<td>Review your source code, following the steps in “How to Upgrade MEX-Files to Use the 64-Bit API” on page 4-38, and then test.</td>
</tr>
</tbody>
</table>

**Must I Update Source MEX-Files on 32-Bit Platforms?**

There are no changes to building 32-bit MEX-files. However, in a future version of MATLAB, the compatibility layer, with the `-compatibleArrayDims` flag, might be unsupported and you then would need to upgrade your MEX-files.

If you build MEX-files exclusively on 32-bit platforms, but want to write platform-independent code, you still can upgrade your code. If possible, build on a 64-bit system to validate your changes.

**What If I Do Not Upgrade?**

On 32-bit platforms, you do not need to make any changes to build MEX-files.

On 64-bit platforms, you can build MEX-files by using the `-compatibleArrayDims` flag.

On 64-bit platforms, if you do not update your source files and you build without the `-compatibleArrayDims` flag, the results are unpredictable. One or more of the following could occur:

- Increased compiler warnings and/or errors from your native compiler
- Run-time errors
- Wrong answers

**How to Upgrade MEX-Files to Use the 64-Bit API**

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 4-39.
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 4-40.
b  “Replace Unsupported Functions” on page 4-42.
c  If necessary, “Update Fortran Source Code” on page 4-44.

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 4-42.

3  Compile using the 64-bit API. To build myMexFile.c, type:

  mex -largeArrayDims myMexFile.c

4  Resolve failures and warnings — see “Resolve -largeArrayDims Build Failures and Warnings” on page 4-43.

5  Compare Results — see “Execute 64-Bit MEX-File and Compare Results with 32-Bit Version” on page 4-43.

6  Check memory — see “Experiment with Large Arrays” on page 4-43.

The following procedures use C/C++ terminology and example code. Fortran MEX-files share the same issues, with more tasks described in “Update Fortran Source Code” on page 4-44.

**Back Up Files and Create Tests**

Before adapting your code to handle large arrays, verify the MEX-file works with the traditional 32-bit array dimensions. At a minimum, build a list of expected inputs and outputs, or create a full test suite. To compare the results with the upgraded source code, use these tests. 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 MX Matrix Library functions — see “Update Arguments Used to Call Functions in the 64-Bit API” on page 4-40.
- Intermediate variables — see “Update Variables Used for Array Indices and Sizes” on page 4-41.
- Variables used as both size/index values and as 32-bit integers — see “Analyze Other Variables” on page 4-41.

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 5-61. 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:

```c
int nrows,ncolumns;
...
y_out = mxCreateDoubleMatrix(nrows, ncolumns, mxREAL);
```

To see the function signature, type:

```bash
doc mxCreateDoubleMatrix
```

The signature is:

```c
mxArray *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.

<table>
<thead>
<tr>
<th>Replace:</th>
<th>With:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int nrows,ncolumns;</td>
<td>mwSize nrows,ncolumns;</td>
</tr>
</tbody>
</table>
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`:

```c
mwSize nrows,ncolumns; /* inputs to mxCreateDoubleMatrix */
int numDataPoints;
```

```c
nrows = 3;
numDataPoints = nrows * 2;
ncolumns = numDataPoints + 1;
```

```c
... 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.

<table>
<thead>
<tr>
<th>Replace:</th>
<th>With:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int numDataPoints;</code></td>
<td><code>mwSize numDataPoints;</code></td>
</tr>
</tbody>
</table>

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.

```c
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:

```c
mxArray *mxCreateDoubleMatrix(mwSize m, mwSize n,```
mxComplexity ComplexFlag)
xArray *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. Replace numSensors with two new variables to handle both functions, as shown in the following table.

<table>
<thead>
<tr>
<th>Replace:</th>
<th>With:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int numSensors;</td>
<td>/* create 2 variables */</td>
</tr>
<tr>
<td></td>
<td>/* of different types */</td>
</tr>
<tr>
<td></td>
<td>mwSize numSensorSize;</td>
</tr>
<tr>
<td></td>
<td>int numSensorFields;</td>
</tr>
<tr>
<td>myNumeric =</td>
<td>/* use mwSize variable */</td>
</tr>
<tr>
<td>mxCreateDoubleMatrix(</td>
<td>/* numSensorSize */</td>
</tr>
<tr>
<td>numSensors,</td>
<td>/* myNumeric = */</td>
</tr>
<tr>
<td>n, mxREAL);</td>
<td>mxCreateDoubleMatrix(</td>
</tr>
<tr>
<td></td>
<td>numSensorSize,</td>
</tr>
<tr>
<td></td>
<td>n, mxREAL);</td>
</tr>
<tr>
<td>myStruct =</td>
<td>/* use int variable */</td>
</tr>
<tr>
<td>mxCreateStructMatrix(</td>
<td>/* numSensorFields */</td>
</tr>
<tr>
<td>m, n,</td>
<td>/* myStruct = */</td>
</tr>
<tr>
<td>numSensors,</td>
<td>mxCreateStructMatrix(</td>
</tr>
<tr>
<td>fieldnames);</td>
<td>m, n,</td>
</tr>
<tr>
<td></td>
<td>numSensorFields,</td>
</tr>
<tr>
<td></td>
<td>fieldnames);</td>
</tr>
</tbody>
</table>

**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 -largeArrayDims 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- and 64-bit types. Refer to http://www.mathworks.com/matlabcentral/answers/99144-how-do-i-update-mex-files-to-use-the-large-array-handling-api-largearraydims, Step 5 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.


After you resolve any 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 mxArrays” on page 5-61.

**Update Fortran Source Code**

All of the previous information applies to Fortran, as well as C/C++. Fortran uses similar API signatures, identical mwSize / mwIndex types, and similar compilers and debuggers. To make your Fortran source code 64-bit compatible, perform these additional tasks:

- “Use Fortran API Header File” on page 4-44
- “Declare Fortran Pointers” on page 4-44
- “Require Fortran Type Declarations” on page 4-44
- “Use Variables in Function Calls” on page 4-45
- “Manage Reduced Fortran Compiler Warnings” on page 4-46

**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 6-2.

**Declare Fortran Pointers**

Pointers are 32- 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 6-5 and mwPointer.

The C/C++ compiler automatically handles pointer size. In Fortran, MATLAB uses the mwPointer type to handle this difference. For example, mxArrayCreateDoubleMatrix returns an mwPointer:

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

**Require Fortran Type Declarations**

Fortran uses implicit type definitions. This means 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:

```fortran
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:

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

Suppose that you have a C/C++ MEX-file with the following statement:

```c
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:

```fortran
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:

```fortran
mwSize nrows, ncols
INTEGER*4 flag
```
nrows = 2
cols = 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 4-43 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 4-43.
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 “Deprecated MEX function” 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.

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Replace <code>mexGet</code> and <code>mexSet</code> Functions” on page 4-47</td>
</tr>
<tr>
<td>“mex Automatically Converts Handle Type” on page 4-50</td>
</tr>
<tr>
<td>“I Want to Rebuild MEX Source Code Files” on page 4-50</td>
</tr>
<tr>
<td>“I Do Not Have MEX Source Code File” on page 4-50</td>
</tr>
</tbody>
</table>

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`.

```c
#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 the “Deprecated MEX function” error.

To change the source file, make the following edits. This code uses the variable plhs[0] in mexGetProperty 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
Upgrade MEX Files to Use Graphics Objects

#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++ 2010 (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 4-50.
**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 the “Deprecated MEX function” error. To rebuild the file, follow the instructions in either “Replace `mexGet` and `mexSet` Functions” on page 4-47 or “I Want to Rebuild MEX Source Code Files” on page 4-50.

**I Want to Rebuild MEX Source Code Files**

If you rebuild your MEX source files in MATLAB R2014b or later, MATLAB displays the “Deprecated MEX function” error.

You might be able to use the `mex` function 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 4-50.

**I Do Not Have MEX Source Code File**

If you get a runtime 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)

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”
Platform Compatibility

In this section...

“Verify the MEX File Is Built for Your Platform” on page 4-52
“Verify Your Architecture on Windows Platforms” on page 4-52

Verify the MEX File Is Built for Your Platform

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 the following table. To determine the extension for your platform, use the `mexext` function.

MEX-File Platform-Dependent Extension

<table>
<thead>
<tr>
<th>Platform</th>
<th>Binary MEX-File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (64-bit)</td>
<td>mexa64</td>
</tr>
<tr>
<td>Apple Mac (64-bit)</td>
<td>mexmaci64</td>
</tr>
<tr>
<td>Microsoft Windows (32-bit)</td>
<td>mexw32</td>
</tr>
<tr>
<td>Windows (64-bit)</td>
<td>mexw64</td>
</tr>
</tbody>
</table>

Verify Your Architecture on Windows Platforms

Verify the MEX file is for the same architecture, 32- vs. 64-bit.

On non-Windows platforms, MATLAB is supported for 64-bit architectures only.

More About

• “Version Compatibility” on page 4-58
Invalid MEX File Error

If MATLAB cannot find all .dll files referenced by a MEX file, it cannot load the MEX file. MATLAB displays the following error message:

Invalid MEX-file `<mexfilename>`:
The specified module could not be found.

where `mexfilename` is the module with the dependency error. This module cannot find its dependent libraries. To resolve this error, find the names of the dependent libraries, and determine if they are present on your system and on the system path.

On Windows systems, to find library dependencies, use the third-party product Dependency Walker. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module 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. Download the Dependency Walker utility from the website http://www.dependencywalker.com/. See http://www.mathworks.com/matlabcentral/answers/92362-how-do-i-determine-which-libraries-my-mex-file-or-stand-alone-application-requires for information on using the Dependency Walker.

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 search path used to find these explicitly loaded libraries is controlled by the code that loads the libraries and 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
- Missing compiler runtime 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 runtime libraries. Contact your MEX file or library vendor.
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 MEX File” on page 4-17 for information about creating the executable function.

If you get runtime errors when you call a MEX file that you did not create, consider the following:

- “Platform Compatibility” on page 4-52
- “Version Compatibility” on page 4-58
- On Windows platforms, install the C++ compiler runtime 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 runtime libraries, 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 of the dependent libraries are available, as expected. However, if you receive a MEX file from another MATLAB user, you might not have all of 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 runtime libraries and language-specific libraries. A MEX file might also use specialized runtime 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 Error” on page 4-53.

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 runtime 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 “Version Compatibility” on page 4-58.
- Language-specific libraries, provided by the compiler. C-language MEX files built on Windows systems require Visual C++ runtime libraries.
- Other runtime 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.

```matlab
% 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.

```matlab
v = ver('matlab');
v.Release
ans =
(R2012a)
```

Identify your platform.

```matlab
archstr = computer('arch')
archstr =
win64
```

Identify the MEX file extension.

```matlab
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 runtime library from the Microsoft Visual C++ 2008 Redistributable Package.
• If you build a MEX file with a third-party library, instructions for acquiring and installing the necessary files.

Related Examples
• “Use Help Files with MEX Files” on page 4-6
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 runtime libraries. MEX files are usually backward compatible, which means you can run a MEX file that was created on an earlier version of MATLAB 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 (forward compatibility), but this is not supported.

More About

• “Platform Compatibility” on page 4-52
Getting Help When MEX Fails

In this section...

“Errors Finding Supported Compiler” on page 4-59
“Errors Building MEX Function” on page 4-59
“Preview mex Build Commands” on page 4-60

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++ 2010 (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:

```
c1 /c /GR /W3 /EHs /nologo /MD /DMX_COMPAT_32 /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE 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`
Understanding MEX File Problems

Use the following figure to help isolate common problems that occur when creating binary MEX files.
Troubleshooting MEX File Creation Problems
In this section...

“Problem 1 — Compiling a Source MEX File Fails” on page 4-63
“Problem 2 — Compiling Your Own Program Fails” on page 4-63
“Problem 3 — Binary MEX File Load Errors” on page 4-64
“Problem 4 — Segmentation Fault” on page 4-65
“Problem 5 — Program Generates Incorrect Results” on page 4-65

Problems 1 through 5 refer to the corresponding numbered sections of the previous flow chart. For additional suggestions on resolving MEX file build problems, see the MathWorks Technical Support website http://www.mathworks.com/support.

Problem 1 — Compiling a Source MEX File Fails

Syntax Errors Compiling C/C++ MEX Files on UNIX

The most common configuration problem in creating C/C++ source MEX files on UNIX systems involves using a non-ANSI C compiler, or failing to pass to the compiler a flag that tells it to compile ANSI C code.

A reliable way of knowing if you have this type of configuration problem is if the header files supplied by MATLAB generate a string of syntax errors when you try to compile your code. See “What You Need to Build MEX Files” on page 4-19 or, if necessary, obtain an ANSI C compiler.

File Not Found on Windows

The mex function cannot find files located in folder names that contain non-ASCII characters.

Problem 2 — Compiling Your Own Program Fails

Mixing ANSI and non-ANSI C code can generate a string of syntax errors. MATLAB provides header and source files that are ANSI C compliant. Therefore, your C code must also be ANSI compliant.

Other common problems that can occur in any C/C++ program are neglecting to include all necessary header files, or neglecting to link against all required libraries.
Make sure that you are using a MATLAB-supported compiler. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website. Additional information can be found in “Compiler- and Platform-Specific Issues” on page 4-66.

**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
```

If you want to call functions from a C/C++ or Fortran library from MATLAB, write a gateway function, as described in “Components of MEX File” on page 5-3.

**Problem 3 — Binary MEX File Load Errors**

If you receive an error of the form:

```
Unable to load mex file:
Invalid MEX-file
```

MATLAB does not recognize your MEX file.

MATLAB loads MEX files by looking for the gateway routine, `mexFunction`. If you misspell the function name, MATLAB cannot load your MEX file and generates an error message. On Windows systems, check that you are exporting `mexFunction` correctly.

On some platforms, if you fail to link against required libraries, you might get an error when MATLAB loads your MEX file rather than when you compile your MEX file. In such cases, a system error message referring to `unresolved symbols` or `unresolved references` appears. Be sure to link against the library that defines the function in question.

On Windows systems, MATLAB fails to load MEX files if it cannot find all .dll files referenced by the MEX file; the .dll files must be on the path or in the same folder as the MEX file. This is also true for third party .dll files. To diagnose this problem, see “Invalid MEX File Error” on page 4-53 for information.
Problem 4 — Segmentation Fault

If a binary MEX-file causes a segmentation violation or assertion, it means the MEX-file attempted to access protected, read-only, or unallocated memory.

These types of programming errors are sometimes 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. Consequently, 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 MX Matrix Library API functions.

To troubleshoot problems of this nature, run MATLAB within a debugging environment. For more information, see “Debugging on Microsoft Windows Platforms” on page 5-52 or “Debug Fortran Source MEX-Files” on page 6-31.

Problem 5 — Program Generates Incorrect Results

If your program generates the wrong answers, there are several causes. First, there could be an error in the computational logic. Second, the program could be reading from an uninitialized section of memory. For example, reading the 11th element of a 10-element vector yields unpredictable results.

Another cause of generating a wrong answer could be overwriting valid data due to memory mishandling. For example, writing to the 15th element of a 10-element vector might overwrite data in the adjacent variable in memory. This case can be handled in a similar manner as segmentation violations, as described in Problem 4.

In all of these cases, you can use mexPrintf to examine data values at intermediate stages. Alternatively, run MATLAB within a debugger.
Compiler- and Platform-Specific Issues

In this section...

“Linux gcc Compiler Version Error” on page 4-66
“Linux gcc -fPIC Errors” on page 4-66

Linux gcc Compiler Version Error


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 in order to create position independent code. For information about using the gcc compiler, see www.gnu.org. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.
Memory Management Issues

Overview

When a MEX file 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 file that is not in plhs. MATLAB also frees any memory that was allocated in the MEX file using the mxMalloc, mxMalloc, or mxRealloc functions.

In general, MathWorks® recommends that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. 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 “Memory Management” on page 5-65 in Creating C/C++ Language MEX Files. 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:

```c
mxArray *temp = mxCreateDoubleMatrix(1,1,mxREAL);
...
 mxFree(temp); /* INCORRECT */
```

**Solution**

Call `mxDestroyArray` instead:

```c
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 MEX file's input arguments. This can cause several strange results, generally having to do with the corruption of the caller's workspace, if the right-hand side argument used is a temporary array (for example, a literal or the result of an expression):

```c
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 `mxSetPr`, `mxSetPi`, `mxSetData`, or `mxSetImagData`, specifying memory that was not allocated by `mxCalloc`, `mxMalloc`, 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); mxSetPr(temp, data);  /* INCORRECT */

**Solution**

Rather than use `mxSetPr` 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 `mxGetPr`:

    mxArray *temp = mxCreateDoubleMatrix(1,5,mxREAL);
    double data[5] = {1,2,3,4,5};
    ...  
    memcpy(mxGetPr(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 mxSetPr, 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);
mxSetPr(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 = mxGetPr(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(mxGetPr(plhs[0]));
mxSetPr(plhs[0], pr);

The first solution is more efficient.

Similar memory leaks can also occur when using mxSetPi, mxSetData, mxSetImagData, mxSetIr, or mxSetJc. 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`.

```c
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:

```c
...  
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” on page 5-65.

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

- “Memory Management” on page 5-65
- “Strategies for Efficient Use of Memory”
- “Automatic Cleanup of Temporary Arrays” on page 5-65
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. The difference between free and fixed source forms is explained in the Fortran Language Reference Manual Source Forms topic. The URL for this topic is:

http://h21007.www2.hp.com/portal/download/files/unprot/Fortran/docs/lrm/lrm0015.htm#source_formatmenu?&Record=383697&STASH=7

The URL for the Fortran Language Reference Manual is:

C/C++ MEX-Files

- “Components of MEX File” on page 5-3
- “MATLAB API Libraries” on page 5-6
- “User Messages” on page 5-8
- “Error Handling” on page 5-9
- “Data Flow in MEX Files” on page 5-10
- “Creating C++ MEX Files” on page 5-13
- “C++ Class in MEX Files” on page 5-15
- “Handle Files with C++” on page 5-16
- “Create C Source MEX File” on page 5-18
- “Table of MEX File Source Code Files” on page 5-24
- “Choose a C++ Compiler” on page 5-28
- “Set Up C/C++ Examples” on page 5-30
- “Pass Scalar Values” on page 5-31
- “Pass Strings” on page 5-34
- “Handling Strings in C/C++” on page 5-36
- “Pass Multiple Inputs or Outputs” on page 5-39
- “Pass Structures and Cell Arrays” on page 5-41
- “Create 2-D Cell Array” on page 5-42
- “Fill mxArray” on page 5-43
- “Prompt User for Input” on page 5-45
- “Handle Complex Data” on page 5-46
- “Handle 8-, 16-, and 32-Bit Data” on page 5-47
- “Manipulate Multidimensional Numerical Arrays” on page 5-48
- “Handle Sparse Arrays” on page 5-50
- “Call MATLAB Functions from C/C++ MEX Files” on page 5-51
• “Debugging on Microsoft Windows Platforms” on page 5-52
• “Debugging on Linux Platforms” on page 5-54
• “Debugging on Mac Platforms” on page 5-56
• “Handling Large mxArrays” on page 5-61
• “Memory Management” on page 5-65
• “Handling Large File I/O” on page 5-68
• “Install MinGW-w64 Compiler” on page 5-74
• “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64” on page 5-77
Components of MEX File

In this section...

“mexFunction Gateway Routine” on page 5-3
“Naming the MEX File” on page 5-3
“Required Parameters” on page 5-3
“Managing Input and Output Parameters” on page 5-4
“Validating Inputs” on page 5-4
“Computational Routine” on page 5-5

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:

```c
void mexFunction(
    int n1hs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
```

Place this function after your computational routine and any other functions in your source file.

The following table describes the parameters for mexFunction.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prhs</td>
<td>Array of right-side input arguments.</td>
</tr>
<tr>
<td>plhs</td>
<td>Array of left-side output arguments.</td>
</tr>
<tr>
<td>nrhs</td>
<td>Number of right-side arguments, or the size of the prhs array.</td>
</tr>
<tr>
<td>nlhs</td>
<td>Number of left-side arguments, or the size of the plhs array.</td>
</tr>
</tbody>
</table>

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:

```c
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 4-68.

### Validating Inputs

For a list of functions to validate inputs to your functions, see the Matrix Library category, “Validate Data”.

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")) {
    mexErrMsgTxt("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**

`mexext` | `mexFunction` | `mxDuplicateArray` | `mxIsClass`

**More About**

- “MATLAB API Libraries” on page 5-6
MATLAB API Libraries

In this section...

“Matrix Library” on page 5-6
“MEX Library” on page 5-6
“Preprocessor Macros” on page 5-6

Use Matrix Library and the MEX Library functions in gateway and computational routines to interact with data in the MATLAB workspace. These libraries 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 Library

Use Matrix Library 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 string 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 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.
See Also
mexEvalString | mexGetVariable | mexPutVariable | mwIndex | mwSize | mxArray

More About
• “MATLAB Data” on page 4-7
• “MEX Library API”
• C/C++ Matrix Library
User Messages

To print a string 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.

For example, the following code snippet prints the input string, `prhs[0]`.

```c
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

The `mexErrMsgIdAndTxt` function prints error information and terminates your binary MEX file. The `mexWarnMsgIdAndTxt` function prints information, but does not terminate the MEX file.

```c
char *buf;
int buflen;

if (mxIsChar(prhs[0])) {
    if (mxGetString(prhs[0], buf, buflen) == 0) {
        mexPrintf("The input string is:  %s\n", buf);
    }
    else {
        mexErrMsgIdAndTxt("MyProg:ConvertString",
                         "Could not convert string data.");
        // exit MEX file
    }
} else {
    mexWarnMsgIdAndTxt("MyProg:InputString",
                      "Input should be a string to print properly.");
}

// continue with processing
```

See Also

`mexErrMsgIdAndTxt` | `mexWarnMsgIdAndTxt`
Data Flow in MEX Files

In this section...

“Showing Data Input and Output” on page 5-10
“Gateway Routine Data Flow Diagram” on page 5-11

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:

```matlab
X = myFunction(Y, Z);
```

The MATLAB interpreter calls `mexFunction`, the gateway routine to `myFunction`, with the following arguments:

- `nlhs = 1`
- `nrhs = 2`

![Gateway Routine Data Flow Diagram]

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 \( \text{nlhs} = 0 \), 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 \([C, D] = \text{func}(A,B)\). 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, `func.c`, 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`. 
MATLAB
A call to
MEX-file func:
\[[C,D]=\text{func}(A,B)\]
tells MATLAB to
pass variables A and
B to your MEX-file.
C and D are left
unassigned.

Inputs
\begin{align*}
\text{const mxArray } & \text{A} \\
\text{A} & = \text{prhs[0]}
\end{align*}

\begin{align*}
\text{const mxArray } & \text{B} \\
\text{B} & = \text{prhs[1]}
\end{align*}

\begin{align*}
\text{void mexFunction(} \\
\text{int } & \text{nlhs, mxArray } \text{plhs[]}, \\
\text{int } & \text{nrhs, const mxArray } \text{prhs[]})
\end{align*}

In the gateway routine:
\begin{itemize}
  \item Use the \text{mxCreate} functions to create
  the MATLAB arrays for your output
  arguments. Set \text{plhs[0]},{1},... to
  the pointers to the newly created
  MATLAB arrays.
  \item Use the \text{mxGet} functions to extract
  your data from \text{prhs[0]},{1},... .
  \item Call your C subroutine passing the
  input and output data pointers as
  function parameters.
\end{itemize}

MATLAB
On return from
MEX-file func:
\[[C,D]=\text{func}(A,B)\]
\text{plhs[0]} is assigned
to \text{C} and \text{plhs[1]} is
assigned to \text{D}.

Outputs
\begin{align*}
\text{mxarray } & \text{D} \\
\text{D} & = \text{plhs[1]}
\end{align*}

\begin{align*}
\text{mxarray } & \text{C} \\
\text{C} & = \text{plhs[0]}
\end{align*}

C/C++ MEX Cycle
Creating C++ MEX Files

In this section...

“Creating Your C++ Source File” on page 5-13
“Compiling and Linking” on page 5-13
“Memory Considerations for Class Destructors” on page 5-13
“Use mexPrintf to Print to MATLAB Command Window” on page 5-14

MEX files support all C++ language standards. This topic discusses specific C++ language issues to consider when creating and using MEX files.

Use the C syntax statements in the MATLAB API libraries in your C++ applications. You can also copy MATLAB C code examples into C++ applications. For example, see the mexcpp.cpp file that contains both C and C++ statements.

Creating Your C++ Source File

The C++ source code for the examples provided by MATLAB 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 file.

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” on page 5-65.
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.

**See Also**

`mexPrintf`

**Related Examples**

- “C++ Class in MEX Files” on page 5-15
- “Handle Files with C++” on page 5-16
- `mexcpp.cpp`

**More About**

- “Build MEX File” on page 4-17
C++ Class in MEX Files

This example, `mexc.cpp`, shows how to use C++ code with your C language MEX file. It uses member functions, constructors, destructors, and the `iostream` include file.

To build this example, at the command prompt type:

```
mex mexcpp.cpp
```

The calling syntax is `mexcpp(num1, num2)`.

The routine 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.

**Related Examples**

- `mexc.cpp`
Handle Files with C++

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 file 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, 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 MEX file, type:

```cpp
mex mexatexit.c
```

Run the example.

```cpp
x = 'my input string';
mexatexit(x)
```

Opening file matlab.data.
Writing data to file.

Clear the MEX file.

```cpp
clear mexatexit
```

Closing file matlab.data.

Display the contents of `matlab.data`.

```cpp
type matlab.data
```

my input string

**See Also**

mexAtExit

**Related Examples**

- mexatexit.cpp
- mexatexit.c
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 matrix. You can use these same C statements in a C++ application.

`arrayProduct` multiplies an n-dimensional array, `y`, by a scalar value, `x`, and returns the results in array, `z`.

```c
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.

```c
/*
 * 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.

```c
#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.

```c
/* 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`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlhs</td>
<td>Number of output (left-side) arguments, or the size of the <code>plhs</code> array.</td>
</tr>
<tr>
<td>plhs</td>
<td>Array of output arguments.</td>
</tr>
<tr>
<td>nrhs</td>
<td>Number of input (right-side) arguments, or the size of the <code>prhs</code> array.</td>
</tr>
<tr>
<td>prhs</td>
<td>Array of input arguments.</td>
</tr>
</tbody>
</table>

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.

```c
if(nrhs != 2) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs",
        "Two inputs required.");
}
```

Use this code to check for one output argument, the product `outMatrix`.

```c
if(nlhs != 1) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs",
        "One output required.");
}
```
Verify the argument types using the \texttt{plhs} and \texttt{prhs} arguments. This code validates that \texttt{multiplier}, represented by \texttt{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 \texttt{inMatrix}, represented by \texttt{prhs[1]}, is type \texttt{double}.

if( !mxIsDouble(prhs[1]) ||
    mxIsComplex(prhs[1])) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notDouble",
                     "Input matrix must be type double.");
}

Validate that \texttt{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.");
}

\textbf{Create Computational Routine}

Add the \texttt{arrayProduct} code. This function is your \textit{computational routine}, the source code that performs the functionality you want to use in MATLAB.

\begin{verbatim}
void arrayProduct(double x, double *y, double *z, int n) {
    int i;

    for (i=0; i<n; i++) {
        z[i] = x * y[i];
    }
}
\end{verbatim}

A computational routine is optional. Alternatively, you can place the code within the \texttt{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.

```c
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.
  ```c
double multiplier;      /* input scalar */
double *inMatrix;       /* 1xN input matrix */
```

- Declare ncols for the size of the input matrix.
  ```c
  mwSize ncols;           /* size of matrix */
  ```

- Declare the output argument, outMatrix.
  ```c
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.

```c
/* get the value of the scalar input */
multiplier = mxGetScalar(prhs[0]);
```

Use the mxGetPr function to point to the input matrix data.

```c
/* create a pointer to the real data in the input matrix */
inMatrix = mxGetPr(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 `mxGetPr` function to assign the `outMatrix` argument to `plhs[0]`

/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(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 in the editor.

**Build Binary MEX File**

At the MATLAB command prompt, build the binary MEX file.

```matlab
mex arrayProduct.c
```

**Test the MEX File**

```matlab
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 file. To test the input variable, `inputArg`, and convert it to `double`, if necessary, use this code.
A = [1.5, 2, 9];
inputArg = int16(A);
if ~strcmp(class(inputArg), 'double')
    inputArg = double(inputArg);
end
B = arrayProduct(s, inputArg)

See Also
mexfunction | mwSize | mxCreateDoubleMatrix | mxGetN | mxGetPr | mxGetScalar

Related Examples
• arrayProduct.c

More About
• “Creating C++ MEX Files” on page 5-13
Table of MEX File Source Code Files

Source code for the MEX examples shown in the following table is in subfolders of `matlabroot/extern/examples`. Make 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:

```matlab
copyfile(fullfile(matlabroot,'extern','examples','foldername','filename'),'.','f')
```

where `filename` is the name of the example, for example `arrayProduct.c`, and `foldername` is the subfolder name, for example `mex`.

You can create and compile MEX files in MATLAB or at your operating system prompt. At either prompt, type:

```matlab
mex -v filename
```

For examples listing multiple Fortran sources files, both files are required to build the MEX file.

```matlab
mex -v file1.F file2.F
```

You can modify the source code for the examples. For convenience, the example instructions open the files in MATLAB Editor, but you can use any code development editor.

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Example Subfolder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayFillGetPr.c</td>
<td>refbook</td>
<td>Fill <code>mxArray</code> using <code>mxGetPr</code>.</td>
</tr>
<tr>
<td>arrayFillSetData.c</td>
<td>refbook</td>
<td>Fill <code>mxArray</code> with non-double values.</td>
</tr>
<tr>
<td>arrayFillSetPr.c</td>
<td>refbook</td>
<td>Fill <code>mxArray</code> using <code>mxSetPr</code> to allocate memory dynamically.</td>
</tr>
<tr>
<td>arrayProduct.c</td>
<td>mex</td>
<td>Multiply a scalar times 1xN matrix.</td>
</tr>
<tr>
<td>arraySize.c</td>
<td>mex</td>
<td>Illustrate memory requirements of large <code>mxArray</code>.</td>
</tr>
<tr>
<td>Example Name</td>
<td>Example Subfolder</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>convec.c</td>
<td>refbook</td>
<td>Pass complex data.</td>
</tr>
<tr>
<td>convec.F</td>
<td>refbook</td>
<td></td>
</tr>
<tr>
<td>dblmat.F</td>
<td>refbook</td>
<td>Use of Fortran %VAL.</td>
</tr>
<tr>
<td>compute.F</td>
<td>refbook</td>
<td></td>
</tr>
<tr>
<td>dotProductComplex.c</td>
<td>refbook</td>
<td>Handle Fortran complex return type for function called from a C MEX file.</td>
</tr>
<tr>
<td>doubleelement.c</td>
<td>refbook</td>
<td>Use unsigned 16-bit integers.</td>
</tr>
<tr>
<td>explore.c</td>
<td>mex</td>
<td>Identify data type of input variable.</td>
</tr>
<tr>
<td>findnz.c</td>
<td>refbook</td>
<td>Use N-dimensional arrays.</td>
</tr>
<tr>
<td>fulltosparse.c</td>
<td>refbook</td>
<td>Populate a sparse matrix.</td>
</tr>
<tr>
<td>fulltosparse.F, loadsparse.F</td>
<td>refbook</td>
<td></td>
</tr>
<tr>
<td>matrixDivide.c</td>
<td>refbook</td>
<td>Call a LAPACK function.</td>
</tr>
<tr>
<td>matrixDivideComplex.c</td>
<td>refbook</td>
<td>Call a LAPACK function with complex numbers.</td>
</tr>
<tr>
<td>matrixMultiply.c</td>
<td>refbook</td>
<td>Call a BLAS function.</td>
</tr>
<tr>
<td>matsq.F</td>
<td>refbook</td>
<td>Pass matrices in Fortran.</td>
</tr>
<tr>
<td>mexatexit.c</td>
<td>mex</td>
<td>Register an exit function to close a data file.</td>
</tr>
<tr>
<td>mexatexit.cpp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mexcallmatlab.c</td>
<td>mex</td>
<td>Call built-in MATLAB disp function.</td>
</tr>
<tr>
<td>mexcallmatlabwithtrap.c</td>
<td>mex</td>
<td>How to capture error information.</td>
</tr>
<tr>
<td>mexcpp.cpp</td>
<td>mex</td>
<td>Illustrate some C++ language features in a MEX file.</td>
</tr>
<tr>
<td>mexevalstring.c</td>
<td>mex</td>
<td>Use mexEvalString to assign variables in MATLAB.</td>
</tr>
<tr>
<td>Example Name</td>
<td>Example Subfolder</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mexfunction.c</td>
<td>mex</td>
<td>How to use <code>mexfunction</code>.</td>
</tr>
<tr>
<td>mexgetproperty.c</td>
<td>mex</td>
<td>Use <code>mxGetProperty</code> and <code>mxSetProperty</code> to change the <code>Color</code> property of a graphic object.</td>
</tr>
<tr>
<td>mexgetarray.c</td>
<td>mex</td>
<td>Use <code>mexGetVariable</code> and <code>mexPutVariable</code> to track counters in the MEX file and in the MATLAB global workspace.</td>
</tr>
<tr>
<td>mexlock.c</td>
<td>mexlockf.F</td>
<td>How to lock and unlock a MEX file.</td>
</tr>
<tr>
<td>mxcalcsinglesubscript.c</td>
<td>mx</td>
<td>Demonstrate MATLAB 1-based matrix indexing versus C 0-based indexing.</td>
</tr>
<tr>
<td>mxcreatecellmatrix.c</td>
<td>mxcreatecellmatrixf.F</td>
<td>Create 2-D cell array.</td>
</tr>
<tr>
<td>mxcreatecharmatrixfromstr.c</td>
<td>mx</td>
<td>Create 2-D string array.</td>
</tr>
<tr>
<td>mxcreatestructarray.c</td>
<td>mx</td>
<td>Create MATLAB structure from C structure.</td>
</tr>
<tr>
<td>mxgeteps.c</td>
<td>mxgetepsf.F</td>
<td>Read MATLAB <code>eps</code> value.</td>
</tr>
<tr>
<td>mxgetinf.c</td>
<td>mx</td>
<td>Read <code>inf</code> value.</td>
</tr>
<tr>
<td>mxgetnzmax.c</td>
<td>mx</td>
<td>Display number of nonzero elements in a sparse matrix and maximum number of nonzero elements it can store.</td>
</tr>
<tr>
<td>mxisclass.c</td>
<td>mx</td>
<td>Check if array is member of specified class.</td>
</tr>
<tr>
<td>mxisfinite.c</td>
<td>mx</td>
<td>Check for NaN and infinite values.</td>
</tr>
<tr>
<td>Example Name</td>
<td>Example Subfolder</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mxislogical.c</td>
<td>mx</td>
<td>Check if workspace variable is logical or global.</td>
</tr>
<tr>
<td>mxmalloc.c</td>
<td>mx</td>
<td>Allocate memory to copy a MATLAB string to a C string.</td>
</tr>
<tr>
<td>mxsetdimensions.c</td>
<td>mx</td>
<td>Reshape an array.</td>
</tr>
<tr>
<td>mxsetdimensionsf.F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mxsetnzmax.c</td>
<td>mx</td>
<td>Reallocate memory for sparse matrix and reset values of pr, pi, ir, and nzmax.</td>
</tr>
<tr>
<td>passstr.F</td>
<td>refbook</td>
<td>Pass C character matrix from Fortran to MATLAB.</td>
</tr>
<tr>
<td>phonebook.c</td>
<td>refbook</td>
<td>Manipulate structures and cell arrays.</td>
</tr>
<tr>
<td>revord.c</td>
<td>refbook</td>
<td>Copy MATLAB string data to and from C-style string.</td>
</tr>
<tr>
<td>revord.F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sincall.c</td>
<td>refbook</td>
<td>Create mxArray and pass to MATLAB sin and plot functions.</td>
</tr>
<tr>
<td>sincall.F, fill.F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>timestwo.c</td>
<td>refbook</td>
<td>Demonstrate common workflow of MEX file.</td>
</tr>
<tr>
<td>timestwo.F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>utdu_slv.c</td>
<td>refbook</td>
<td>Use LAPACK for symmetric indefinite factorization.</td>
</tr>
<tr>
<td>xtimesy.c</td>
<td>refbook</td>
<td>Pass multiple parameters.</td>
</tr>
<tr>
<td>xtimesy.F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yprime.c</td>
<td>mex</td>
<td>Solve simple three body orbit problem.</td>
</tr>
<tr>
<td>yprimef.F, yprimefg.F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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++ 2010. 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 2010, click the link:

Microsoft Visual C++ 2010

MEX configured to use 'Microsoft Visual C++ 2010' for C++ language compilation.

Microsoft Visual C++ 2010 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.

**More About**

- “Change Default Compiler” on page 4-20
Set Up C/C++ Examples

The Matrix Library provides a full set of 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. The first example discusses the simple case of doubling a scalar. After that, the examples discuss how to pass in, manipulate, and pass back various data types, and how to handle multiple inputs and outputs. Finally, the sections discuss passing and manipulating various MATLAB types.

Source code for the examples in this section is in the `matlabroot/extern/examples/refbook` folder. To build an example, first copy the file to a writable folder on your path:

```matlab
copyfile(fullfile(matlabroot,'extern','examples','refbook','filename.c'),'.','f')
```

where `filename` is the name of the example.

At the MATLAB command prompt, type:

```matlab
mex filename.c
```

The following topics look 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 “Table of MEX File Source Code Files” on page 5-24.
Pass Scalar Values

In this section...

“Pass Scalar as Matrix” on page 5-31
“Pass Scalar by Value” on page 5-32

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.

```c
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 the file, 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.

```c
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 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 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 the file, *timestwoalt.c*, in the MATLAB Editor.

Compare the call to *timestwo_alt* to the call to *timestwo*.

```c
/* 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 = mxGetPr(plhs[0]);

/* Call the timestwo_alt subroutine */
timestwo_alt(y,x);

/* Assign pointers to each input and output. */
x = mxGetPr(prhs[0]);
y = mxGetPr(plhs[0]);

/* Call the timestwo subroutine. */
timestwo(y,x);
```

The value *x*, created by *mxGetScalar*, is a scalar not a pointer.
Pass Strings

This example shows how to pass strings to a MEX file. The example `revord.c` accepts a string and returns the characters in reverse order.

C Code Analysis

To see the code, open the file, `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`.

```c
input_buf = mxArrayToString(prhs[0]);
```

Allocate memory for the output argument, `output_buf`, a C-style string.

```c
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`.

```c
revord(input_buf, buflen, output_buf);
```

Convert the output, `output_buf`, to an `mxArray` and assign to `plhs[0]`.

```c
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.

```c
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.

```matlab
x = 'hello world';
y = revord(x)
```

```matlab
y =
dlrow olleh
```

**Related Examples**

- revord.c
Handling Strings in C/C++

In this section...

“How MATLAB Represents Strings in MEX-Files” on page 5-36
“How MATLAB Represents Strings in MEX-Files” on page 5-36
“Character Encoding and Multibyte Encoding Schemes” on page 5-36
“How MATLAB Represents Strings in MEX-Files” on page 5-36
“Converting MATLAB String to C-Style String” on page 5-37
“How MATLAB Represents Strings in MEX-Files” on page 5-36
“Converting C-Style String to MATLAB String” on page 5-37
“How MATLAB Represents Strings in MEX-Files” on page 5-36
“Returning Modified Input String” on page 5-37
“How MATLAB Represents Strings in MEX-Files” on page 5-36
“Memory Management” on page 5-37

How MATLAB Represents Strings in MEX-Files

In C/C++ MEX-files, a MATLAB string 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/C++ Matrix Library functions provides string handling functions to help
you work with both mxArrays and C-style strings.

• mxCreateString — Creates a string mxArray initialized to the input string.
• mxArrayToString — Copies a string mxArray into a C-style string. Supports
  multibyte encoded characters.
• mxArrayGetString — Copies a string mxArray into a C-style string. Best used with single-
  byte encoded characters. Supports multibyte encoded characters when you calculate
  string buffer size.
• mxArrayGetChars — 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 String 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 String

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 your MEX-file takes a string input argument, 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:
| • 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).
xmCreateString | mxDestroyArray
mxGetChars | None. Function creates a pointer to an mxArray but does not allocate additional memory.

More About
- “C/C++ Matrix Library API”
- “Locale Settings for MATLAB Process”
Pass Multiple Inputs or Outputs

This example shows how to call a MEX file with multiple inputs. The function, `xtimesy`, multiplies an input scalar by a scalar or matrix and outputs a matrix.

The `plhs[]` and `prhs[]` parameters are vectors that contain pointers to each left-side (output) variable and each right-side (input) variable, respectively. `plhs[0]` contains a pointer to the first left-side argument, `plhs[1]` contains a pointer to the second left-side argument, and so on. Likewise, `prhs[0]` contains a pointer to the first right-side argument, `prhs[1]` points to the second, and so on.

**C Code Analysis**

To see the code, open the file, `xtimesy.c`, in the MATLAB Editor.

Get the scalar input value, `x`.

```c
x = mxGetScalar(prhs[0]);
```

Get the second input, `y`, which can be either a scalar or a matrix.

```c
y = mxGetPr(prhs[1]);
```

Create the output argument, `plhs[0]`, which is the same size as argument `y`.

```c
/* get the dimensions of the matrix input y */
mrows = mxGetM(prhs[1]);
ncols = mxGetN(prhs[1]);

/* set the output pointer to the output matrix */
plhs[0] = mxCreateDoubleMatrix((mwSize)mrows, (mwSize)ncols, mxREAL);
```

**Build and Test Example**

Run the following commands from the MATLAB command line.

Build the example.

```bash
mex -v xtimesy.c
```

Call the function with scalar values.

```matlab
x = 7;
y = 7;
```
\[
z = \text{xtimesy}(x,y)
\]
\[
z = 49
\]

Call the function with a matrix.

\[
x = 9;
y = \text{ones}(3);
z = \text{xtimesy}(x,y)
\]
\[
z =
\begin{bmatrix}
9 & 9 & 9 \\ 9 & 9 & 9 \\ 9 & 9 & 9 
\end{bmatrix}
\]

**Related Examples**

- `xtimesy.c`
Pass Structures and Cell Arrays

Passing structures and cell arrays into MEX files is like passing any other data type, except the data itself 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/C++ routine, use an API function such as `mxGetData` to access it.

This example takes an \( m \times n \) structure matrix as input and returns a new 1-by-1 structure that contains these fields:

- String input generates an \( m \times n \) cell array
- Numeric input (noncomplex, scalar values) generates an \( m \times n \) 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, enter this structure:

```matlab
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]
```

**Related Examples**

- `phonebook.c`
Create 2-D Cell Array

This example shows how to create a cell array in a MEX file, using the `mxcreatecellmatrix.c` function, which places input arguments in a cell array.

**C Code Analysis**

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

Create a cell array for the number of input arguments.

```c
cell_array_ptr = mxCreateCellMatrix((mwSize)nrhs,1);
```

Copy the input arguments into the cell array.

```c
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.

```bash
mex -v mxcreatecellmatrix.c
```

Create input arguments.

```c
str1 = 'hello';
str2 = 'world';
num = 2012;
```

Create a 3-x-1 cell array and call `disp` to display the contents.

```c
mxcreatecellmatrix(str1,str2,num)
```

The contents of the created cell is:

```matlab
    'hello'
    'world'
    [2012]
```

**Related Examples**

- `mxcreatecellmatrix.c`
Fill mxArray

In this section...

“Options” on page 5-43
“Copying Data Directly into an mxArray” on page 5-43
“Pointing to Data” on page 5-43

Options

You can move data from a C/C++ program into an mxArray using the Matrix Library. The functions you use depend on the type of data in your application. Use the mxSetPr and mxGetPr functions for data of type double. For numeric data other than double, use the mxSetData function. 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].

These examples use real data only. If you have complex data, use the mxGetPi and mxSetPi functions as needed.

Copying Data Directly into an mxArray

The arrayFillGetPr.c example uses the mxGetPr function to copy the values from data to plhs[0].

Pointing to Data

The arrayFillSetPr.c example uses the mxSetPr function to point plhs[0] to data.

The example arrayFillSetData.c shows how to fill an mxArray for numeric types other than double.

See Also
mxCreateString | mxGetPr | mxSetData | mxSetPr
Related Examples

- arrayFillGetPr.c
- arrayFillSetPr.c
- arrayFillSetData.c
Prompt User for Input

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.

```c
#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
Handle Complex Data

MATLAB separates complex data into real and imaginary parts. The MATLAB API provides two functions, `mxGetPr` and `mxGetPi`, that return pointers (of type `double *`) to the real and imaginary parts of your data.

This example, `convec.c`, takes two complex row vectors and convolves them.

To build this example, at the command prompt type:

```bash
mex convec.c
```

Enter these numbers at the MATLAB prompt.

```matlab
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];
```

Call the MEX file.

```matlab
z = convec(x,y)
```

```matlab
z =
    1.0e+02 *
  0.1800 - 0.2600i  0.9600 + 0.2800i  1.3200 - 1.4400i  3.7600 - 0.1200i
  1.5400 - 4.1400i
```

Compare the results with the built-in MATLAB function `conv`.

**See Also**

`mxGetPi` | `mxGetPr`

**Related Examples**

- `convec.c`
Handle 8-, 16-, and 32-Bit Data

The MATLAB API provides a set of functions that support signed and unsigned 8-, 16-, and 32-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 `mxGetData` and `mxGetImagData`. These two functions return pointers to the real and imaginary data. You can perform arithmetic on data of 8-, 16-, or 32-bit precision in MEX files and return the result to MATLAB, which recognizes the correct data class.

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:

```bash
mex doubleelement.c
```

Call the example.

```c
doubleelement
```

```plaintext
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` | `mxGetData` | `mxGetImagData`

Related Examples

- `doubleelement.c`
Manipulate Multidimensional Numerical Arrays

You can manipulate multidimensional numerical arrays by using `mxGetData` and `mxGetImagData`. These functions return pointers to the real and imaginary parts of the data stored in the original multidimensional array. 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 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.

```
findnz(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
mxGetData | mxGetImagData

Related Examples
• findnz.c
Handle Sparse Arrays

The MATLAB 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 \( \text{ir} \) and \( \text{jc} \), two of the parameters associated with sparse arrays. For more information on how MATLAB stores sparse arrays, see “The MATLAB Array” on page 4-7.

The example `fulltospars.c` shows how to populate a sparse matrix.

Build the example.

```c
mex fulltospars.c
```

Create a full, 5-by-5 identity matrix.

```c
full = eye(5)
```

```c
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 `fulltospars` to produce the corresponding sparse matrix.

```c
spar = fulltospars(full)
```

```c
spar =
  (1,1)  1
  (2,2)  1
  (3,3)  1
  (4,4)  1
  (5,5)  1
```

Related Examples

- `fulltospars.c`
Call MATLAB Functions from C/C++ MEX Files

It is possible to call MATLAB functions, operators, user-defined functions, and other binary MEX files from within your C/C++ source code by using the API function mexCallMATLAB.

The example, sincall.c, 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 build this example, at the command prompt type:

mex sincall.c

Run the example.
sincall

MATLAB displays a sin curve equivalent to executing the following MATLAB commands:

```
MAX = 1000;
mm = MAX/2;
for i = 1:mm-1
    X(i) = i*(4*3.14159/MAX);
end
Y = sin(X);
plot(X,Y)
```

See Also
mexCallMATLAB

Related Examples
- sincall.c
Debugging on Microsoft Windows Platforms

This example shows how to debug yprime.c, found in your `matlabroot/extern/examples/mex/` folder, with Microsoft Visual Studio 2010.

1. Make sure Visual Studio is your selected C compiler:
   ```matlab
   cc = mex.getCompilerConfigurations('C','Selected');
   cc.Name
   ans =
   Microsoft Visual C++ 2010 (C)
   ``

2. Compile the source MEX file with the `-g` option, which builds the file with debugging symbols included. For example:
   ```matlab
   copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'),'.','f')
   mex -g yprime.c
   ``


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:
   ```matlab
   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 “Troubleshoot MEX Files”.
Debugging 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` function `-D` option.

   `linux> matlab -Dgdb`

3. Tell gdb to stop for debugging.

   `<gdb> handle SIGSEGV SIGBUS nostop noprint`

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.
Debugging on Mac Platforms

In this section...

“Using Xcode” on page 5-56
“Using LLDB” on page 5-58

Using Xcode

This example shows how to debug the MEX-file, yprime.c, using Xcode.

Copy the 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 /usr/work/my_data. Create the folder if it does not already exist.

```matlab
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'), ...
fullfile('/','usr','work','my_data'))
```

Set your current folder in MATLAB to /usr/work/my_data.

Compile the source MEX file

Compile the source MEX-file with the -g option, which adds debugging symbols.

```matlab
mex -g yprime.c
```

MATLAB creates the binary MEX-file, yprime.mexmaci64.

Create an empty Xcode project for debugging

In Xcode,

• Select File > New > Project.
• In the Choose a template for your project dialog box, in the OS X section, select Other.
• Select Empty.
• Set Product Name to debug_yprime.
Add yprime files to the project

To add breakpoints to your source code file, add the yprime.c file to the project. You can either drag the file directly into the project or right-click in the project and click Add files to "debug_yprime" to add files.

Make sure the Destination option, Copy items into destination group's folder (if needed), is unchecked. Unchecking this option enables breakpoints to be added to the file that MATLAB runs.

Click Finish to add the file.

Create a scheme

- Select Product > Scheme > New Scheme....
- Set Name to debug.
- Set Target to None.
- Press OK. The scheme editing dialog box opens.
- Set the Run > Info > Executable option to the MATLAB executable to use to debug the MEX-file, for example, MATLAB_R2014a.app.

Add a symbolic breakpoint

Open the Debug menu:

- In Xcode 4.6.x:
  Select Product > Debug > Create Symbolic Breakpoint .
- In Xcode 5.0 and later:
  Select Debug > Breakpoints > Create Symbolic Breakpoint.

Set Symbol to NSApplicationMain.

Click Add action to add the following debugger command:
process handle -p true -n false -s false SIGSEGV SIGBUS

Check Automatically continue after evaluating.

Set breakpoints

To add a breakpoint to yprime.c, click the gutter next to the line where you want execution to pause. For more information, refer to the Xcode documentation.
Start MATLAB

Click Run at the top left of the project window (or type Command-R) to start the MATLAB executable. The executable pauses twice; press Continue.

Run the binary MEX-file in MATLAB

In MATLAB, change the current folder to the folder with the yprime files.

Run the binary MEX-file.

yprime(1,1:4)

The debugger opens yprime.c at the first breakpoint.

Press Continue. MATLAB displays:

ans =

2.0000 8.9685 4.0000 -1.0947

Using LLDB

LLDB is the debugger available with Xcode on Mac OS X 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” on page 5-58
• “Debug MEX with JVM” on page 5-60

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 llmdb 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.
Handling Large mxArrays

In this section...

“Using the 64-Bit API” on page 5-61
“Building the Binary MEX File” on page 5-63
“Example” on page 5-63
“Caution Using Negative Values” on page 5-63
“Building Cross-Platform Applications” on page 5-64

Binary MEX files built on 64-bit platforms can handle 64-bit mxArrays. 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” in the Programming Fundamentals documentation. Memory management within source MEX files can have special considerations, as described in “Memory Management” on page 5-65.

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.

<table>
<thead>
<tr>
<th>mxArray Functions Using mwSize/mwIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxCalcSingleSubscript</td>
</tr>
<tr>
<td>mxAlloc</td>
</tr>
<tr>
<td>mxCopyCharacterToPoint</td>
</tr>
<tr>
<td>mxCreateSparseLogicalMatrix</td>
</tr>
<tr>
<td>mxCreateStructArray</td>
</tr>
<tr>
<td>mxCreateStructMatrix</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td><code>mxCopyComplex16ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyComplex8ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyInteger1ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyInteger2ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyInteger4ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToCharacter</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToComplex16</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToComplex8</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToInteger1</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToInteger2</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToInteger4</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToPtrArray</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToReal4</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToReal8</code></td>
</tr>
<tr>
<td><code>mxCreateCellArray</code></td>
</tr>
<tr>
<td><code>mxCreateCellMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateCharArray</code></td>
</tr>
<tr>
<td><code>mxCreateCharMatrixFromStrings</code></td>
</tr>
<tr>
<td><code>mxCreateDoubleMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateLogicalArray</code></td>
</tr>
<tr>
<td><code>mxCreateLogicalMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateNumericArray</code></td>
</tr>
<tr>
<td><code>mxCreateNumericMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateSparse</code></td>
</tr>
<tr>
<td><code>mxCopyComplex16ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyComplex8ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyInteger1ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyInteger2ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyInteger4ToPtr</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToCharacter</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToComplex16</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToComplex8</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToInteger1</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToInteger2</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToInteger4</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToPtrArray</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToReal4</code></td>
</tr>
<tr>
<td><code>mxCopyPtrToReal8</code></td>
</tr>
<tr>
<td><code>mxCreateCellArray</code></td>
</tr>
<tr>
<td><code>mxCreateCellMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateCharArray</code></td>
</tr>
<tr>
<td><code>mxCreateCharMatrixFromStrings</code></td>
</tr>
<tr>
<td><code>mxCreateDoubleMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateLogicalArray</code></td>
</tr>
<tr>
<td><code>mxCreateLogicalMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateNumericArray</code></td>
</tr>
<tr>
<td><code>mxCreateNumericMatrix</code></td>
</tr>
<tr>
<td><code>mxCreateSparse</code></td>
</tr>
</tbody>
</table>

\(^1\) Fortran function only.
C function only.

Functions in this API use the `mwIndex` and `mwSize` types.

### Building the Binary MEX File

Use the `mex` build script option `-largeArrayDims` with the 64-bit API.

### Example

The example, `arraySize.c` in `matlabroot/extern/examples/mex`, shows memory requirements of large `mxArray`s. To see the example, open the file in 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.

```bash
mex -largeArrayDims arraySize.c
```

Run the MEX file.

```c
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- 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.
Automatic Cleanup of Temporary Arrays

When a MEX file returns control to MATLAB, it returns the results of its computations in the output arguments—the `mxArray`s contained in the left-side arguments `plhs[]`. These arrays must have a temporary scope, so do not pass arrays created with the `mxMakeArrayPersistent` function in `plhs`. MATLAB destroys any `mxArray` created by the MEX file that is not in `plhs`. MATLAB also frees any memory that was allocated in the MEX file using the `mxCalloc`, `mxMalloc`, or `mxRealloc` functions.

MathWorks recommends that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. However, there are several circumstances in which the MEX file does not reach its normal return statement.

The normal return is not reached if:

- MATLAB calls `mexErrMsgTxt`.
- 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 file execution using `Ctrl+C`.
- The binary MEX file runs out of memory. The MATLAB out-of-memory handler terminates the MEX file.

For the first two cases, a MEX file 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 `mxCalloc` 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 file. For example, if the first input to your function (prhs[0]) is a string, in order to manipulate the string, create a buffer buf of size buflen. The following statements declare these variables:

```c
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:

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

Next, allocate memory for buf:

```c
buf = mxMalloc(buflen);
```

At the end of the program, if you do not return buf as a p1hs output parameter, then free its memory as follows:

```c
mxFree(buf);
```

Before exiting the MEX file, destroy any temporary arrays and free any dynamically allocated memory, except if such an mxArray is returned in the output argument list, returned by mexGetVariablePtr, 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.

Persistent Arrays

You can exempt an array, or a piece of memory, from the MATLAB automatic cleanup by calling `mexMakeArrayPersistent` or `mexMakeMemoryPersistent`. However, if a MEX file creates such persistent objects, there is a danger that a memory leak could occur if the MEX file 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.

For example, here is a source MEX file that 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. */
        *mxGetPr(persistent_array_ptr) = 1.0;
    } else {
        mexPrintf("MEX file executing; value of first array element is %g\n",
            *mxGetPr(persistent_array_ptr));
    }
}

See Also
mexAtExit | mexMakeArrayPersistent | mxCalloc | mxFree

More About
- “Memory Allocation”
- “Memory Management Issues” on page 4-67
Handling Large File I/O

In this section...

- “Prerequisites to Using 64-Bit I/O” on page 5-68
- “Specifying Constant Literal Values” on page 5-70
- “Opening a File” on page 5-70
- “Printing Formatted Messages” on page 5-71
- “Replacing fseek and ftell with 64-Bit Functions” on page 5-71
- “Determining the Size of an Open File” on page 5-72
- “Determining the Size of a Closed File” on page 5-73

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 file programs.

- “Header File” on page 5-68
- “Type Declarations” on page 5-68
- “Functions” on page 5-69

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:

```c
#include "io64.h"
#include "mex.h"
```

Type Declarations

To declare variables used in 64-bit file I/O, use the following types.

<table>
<thead>
<tr>
<th>MEX Type</th>
<th>Description</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fpos_T</code></td>
<td>Declares a 64-bit int type for <code>setFilePos()</code> and</td>
<td><code>fpos_t</code></td>
</tr>
<tr>
<td>MEX Type</td>
<td>Description</td>
<td>POSIX</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>getFilePos()</td>
<td>Defined in io64.h.</td>
<td></td>
</tr>
<tr>
<td>int64_T, uint64_T</td>
<td>Declares 64-bit signed and unsigned integer types. Defined in tmwtypes.h.</td>
<td>long, long</td>
</tr>
<tr>
<td>structStat</td>
<td>Declares a structure to hold the size of a file. Defined in io64.h.</td>
<td>struct stat</td>
</tr>
<tr>
<td>FMT64</td>
<td>Used in mexPrintf to specify length within a format specifier such as %d.</td>
<td>%lld</td>
</tr>
<tr>
<td>LL, LLU</td>
<td>Suffixes for literal int constant 64-bit values (C Standard ISO®/IEC 9899:1999(E) Section 6.4.4.1). Used only on UNIX systems.</td>
<td>LL, LLU</td>
</tr>
</tbody>
</table>

**Functions**

Use the following functions for 64-bit file I/O. All are defined in the header file io64.h.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileno()</td>
<td>Gets a file descriptor from a file pointer</td>
<td>fileno()</td>
</tr>
<tr>
<td>fopen()</td>
<td>Opens the file and obtains the file pointer</td>
<td>fopen()</td>
</tr>
<tr>
<td>getFileFstat()</td>
<td>Gets the file size of a given file pointer</td>
<td>fstat()</td>
</tr>
<tr>
<td>getFilePos()</td>
<td>Gets the file position for the next I/O</td>
<td>fgetpos()</td>
</tr>
<tr>
<td>getFileStat()</td>
<td>Gets the file size of a given file name</td>
<td>stat()</td>
</tr>
<tr>
<td>setFilePos()</td>
<td>Sets the file position for the next I/O</td>
<td>fsetpos()</td>
</tr>
</tbody>
</table>
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 $2 \times 2^{31}-1$, 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:

```c
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                     const mxArray *prhs[])
{
    #if defined(_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.

```c
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);
      mexErrMsgTxt(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:

```c
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.

```c
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.

```c
getFilePos( fp, (fpos_T*) &position);
setFilePos( fp, (fpos_T*) &position);
```
Getting the File Size

The `getFileSizeStat` function takes a file descriptor input argument. Use `fileno` function to get the file pointer of the open file. `getFileSizeStat` returns the size of that file in bytes in the `st_size` field of a `structStat` structure.

```c
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileSizeStat(fileno(fp), &statbuf))
{
    fileSize = statbuf.st_size;
    mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
}
```

Determining the Size of a Closed File

The `getFileSizeStat` function takes the file name of a closed file as an input argument. `getFileSizeStat` returns the size of the file in bytes in the `st_size` field of a `structStat` structure.

```c
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileSizeStat(filename, &statbuf))
{
    fileSize = statbuf.st_size;
    mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
}
```
Install MinGW-w64 Compiler

You can use the MinGW-w64 version 4.9.2 compiler from TDM-GCC to build MEX files and standalone MATLAB engine and MAT-file applications.

Installing Compiler from Add-Ons Menu

To install the MinGW compiler, open the installer from MATLAB Add-Ons menu. On the Home tab, in the Environment section, click the Add-Ons icon. Under Refine by Type, select Features, or search for MinGW. Follow the prompts to add the compiler.

Note: To install the MATLAB-supported version of MinGW-w64, you must clear the check box highlighted in the following figure.
Install MinGW-w64 Compiler

For more information, see “Support Package Installation” on page 17-2.

**Building yprime.c Example**

You can test the MinGW compiler by building the `yprime.c` example. Copy the source file to a writable folder.

```matlab
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.

```matlab
mex -setup
```

Build the MEX file.

```matlab
mex yprime.c
```

MATLAB displays a “Building with” message showing what compiler was used to build the MEX file.

Run the function.

```matlab
yprime(1,1:4)
```

For more information, see “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64” on page 5-77.

**MinGW Installation Folder Cannot Contain Space**

Do not install MinGW in a location with spaces in the path name. For example, do not use:

```matlab
C:\Program Files\TDM-GCC-64
```

Instead, use:

```matlab
C:\TDM-GCC-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 in order 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.

**More About**
• “Support Package Installation” on page 17-2
• “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64” on page 5-77

**External Websites**
• Supported and Compatible Compilers
Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64

In this section...

“Do Not Link to Library Files Compiled with Non-MinGW Compilers” on page 5-77
“MinGW Compiler Not Setup for Use with MEX” on page 5-77
“MinGW Installation Folder Cannot Contain Space” on page 5-78
“MEX Command Does not Choose MinGW” on page 5-78
“Manually Configure MinGW for MATLAB” on page 5-78
“Potential Memory Leak Inside C++ MEX Files on Using MEX Exceptions” on page 5-79
“Unhandled Explicit Exceptions in C++ MEX Files Unexpectedly Terminate MATLAB” on page 5-80
“Out of Memory Error for Variables Containing Large Amounts of Data” on page 5-81

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 Compiler Not Setup for Use with MEX

If you installed the MinGW compiler but MATLAB does not recognize it, the compiler might not be the supported version.

To install the MATLAB-supported version of MinGW-w64, you must clear the Check for updated files on the TDM-GCC server check box. This check box appears during TDM-GCC Setup when you are asked to Choose which action you want the setup wizard to perform. For information about installing MinGW, see “Installing Compiler from Add-Ons Menu” on page 5-74.
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\TDM-GCC-64

Instead, use:

C:\TDM-GCC-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

MATLAB detects the MinGW compiler by reading an environment variable, `MW_MINGW64_LOC`. When you install MinGW from the MATLAB Add-Ons menu, MATLAB sets this variable.

If necessary, you can set this variable globally, if you have Windows administrative privileges, using the Windows Control Panel. Alternatively, set the variable every time you open MATLAB.

Note: Verify you have installed MinGW-w64 version 4.9.2 before setting the environment variable.

Set variable using Windows Control Panel

To set the environment variable on Windows 7:
Make sure you have administrative privileges.

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 **New**.

In the **New System Variable** dialog box, type MW_MINGW64_LOC in the **Variable name** field.

In the **Variable value** field, type the location of the MinGW-w64 compiler installation, for example, C:\TDM-GCC-64.

Click **Ok** to close the dialog boxes, then close the **Control Panel** dialog box.

**Set variable for MATLAB session**

To set the environment variable in MATLAB, type:

```matlab
setenv('MW_MINGW64_LOC', 'folder')
```

where `folder` is the installation directory, for example, 'C:\TDM-GCC-64'. You must set the variable every time you run MATLAB.

**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.

```cpp
#include "mex.h"

class MyClass {
    public:
        MyClass() {
            mexPrintf("Constructor called");
        }
        ~MyClass() {
            mexPrintf("Destructor called");
        }
};
```
C/C++ MEX-Files

```
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 \texttt{MyClass}, then checks the number of input arguments. If the MEX function calls \texttt{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 \texttt{catch} statement, then the exception causes MATLAB to terminate instead of propagating the error to the MATLAB command line.

```cpp
#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.

Work around

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
    }
}

Out of Memory Error for Variables Containing Large Amounts of Data

The TDM-GCC MinGW 4.9.2 compiler cannot allocate memory for variables in MEX files that contain large amounts of data. The compiler displays an error message like:

out of memory allocating XXXXXX bytes

See Also
mexErrMsgIdAndTxt
Fortran MEX-Files

- “Components of Fortran MEX File” on page 6-2
- “MATLAB Fortran API Libraries” on page 6-5
- “Data Flow in Fortran MEX Files” on page 6-8
- “User Messages” on page 6-11
- “Error Handling” on page 6-12
- “Build Fortran MEX File” on page 6-13
- “Create Fortran Source MEX File” on page 6-14
- “Set Up Fortran Examples” on page 6-19
- “Pass Scalar Values” on page 6-20
- “Pass Strings” on page 6-21
- “Pass Arrays of Strings” on page 6-22
- “Pass Matrices” on page 6-23
- “Pass Integers” on page 6-24
- “Pass Multiple Inputs or Outputs” on page 6-25
- “Handle Complex Data” on page 6-26
- “Dynamically Allocate Memory” on page 6-27
- “Handle Sparse Matrices” on page 6-28
- “Call MATLAB Functions from Fortran MEX Files” on page 6-29
- “Debug Fortran Source MEX-Files” on page 6-31
- “Handling Large mxArrays” on page 6-34
- “Memory Management” on page 6-37
- “MATLAB Supports Fortran 77” on page 6-38
Components of Fortran MEX File

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 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:

```fortran
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`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prhs</code></td>
<td>Array of right-side input arguments.</td>
</tr>
<tr>
<td><code>plhs</code></td>
<td>Array of left-side output arguments.</td>
</tr>
<tr>
<td><code>nrhs</code></td>
<td>Number of right-side arguments, or the size of the <code>prhs</code> array.</td>
</tr>
<tr>
<td><code>nlhs</code></td>
<td>Number of left-side arguments, or the size of the <code>plhs</code> array.</td>
</tr>
</tbody>
</table>

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 of copy of the input array. For example:

```fortran
plhs(1) = mxDuplicateArray(prhs(1))
```

For more information, see the troubleshooting topic “Incorrectly Constructing a Cell or Structure `mxArray`” on page 4-68.
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

`mexext` | `mexFunction` | `mwPointer` | `mxDuplicateArray` | `mxIsClass`
The Matrix Library and the MEX Library describe functions you can use in your gateway and computational routines that interact with MATLAB programs and the data in the MATLAB workspace. These libraries are part of the MATLAB C/C++ and Fortran API Reference library.

To use these functions, include the fintrf header, which declares the entry point and interface routines. Put this statement in your source file:

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

**Matrix Library**

Use Fortran Matrix Library 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. 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. \texttt{mwIndex} represents index values, such as indices into arrays.

MATLAB has an extra preprocessor macro for Fortran files, \texttt{mwPointer}. MATLAB uses a unique data type, the \texttt{mxArray}. Because you cannot create a data type in Fortran, MATLAB passes a special identifier, created by the \texttt{mwPointer} preprocessor macro, to a Fortran program. This is how you get information about an \texttt{mxArray} in a native Fortran data type. For example, you can find out the size of the \texttt{mxArray}, determine whether it is a string, and look at the contents of the array. Use \texttt{mwPointer} to build platform-independent code.

The Fortran preprocessor converts \texttt{mwPointer} to \texttt{integer*4} when building binary MEX files on 32-bit platforms and to \texttt{integer*8} when building on 64-bit platforms.

\textbf{Note:} Declaring a pointer to be the incorrect size might cause your program to crash.

\section*{Using the Fortran \%val\_1 Construct}

The Fortran \texttt{\%val(arg)} construct specifies that an argument, \texttt{arg}, is to be passed by value, instead of by reference. Most, but not all, Fortran compilers support the \texttt{\%val} construct.

If your compiler does not support the \texttt{\%val} construct, copy the array values into a temporary true Fortran array using the \texttt{mxCopy\_*} routines (for example, \texttt{mxCopyPtrToReal8}).

\texttt{\%val\_1 Construct Example}

If your compiler supports the \texttt{\%val} construct, you can use routines that point directly to the data (that is, the pointer returned by \texttt{mxGetPr} or \texttt{mxGetPi}). You can use \texttt{\%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, \texttt{yprime}, by:

\begin{verbatim}
call yprime(%val(yp), %val(t), %val(y))
\end{verbatim}

If your Fortran compiler does not support the \texttt{\%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 Library API” | “MEX Library API” | mwIndex | mwPointer | mwSize | mxArray
Data Flow in Fortran MEX Files

In this section...

“Showing Data Input and Output” on page 6-8
“Gateway Routine Data Flow Diagram” on page 6-9

Showing Data Input and Output

Suppose your MEX-file myFunction has two input arguments and one output argument. The MATLAB syntax is \[ \text{[X]} = \text{myFunction}(Y, Z) \]. To call myFunction from MATLAB, type:

\[
X = \text{myFunction}(Y, Z);
\]

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 $\text{nlhs} = 0$, 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 $\text{func}$ is $[C, D] = \text{func}(A,B)$. In the figure, a call to $\text{func}$ tells MATLAB to pass variables $A$ and $B$ to your MEX-file. $C$ and $D$ are left unassigned.

The gateway routine uses the $\text{mxCreate*}$ functions to create the MATLAB arrays for your output arguments. It sets $\text{plhs}[0]$ and $\text{plhs}[1]$ to the pointers to the newly created MATLAB arrays. It uses the $\text{mxGet*}$ functions to extract your data from your input arguments $\text{prhs}[0]$ and $\text{prhs}[1]$. Finally, it calls your computational routine, passing the input and output data pointers as function parameters.

MATLAB assigns $\text{plhs}[0]$ to $C$ and $\text{plhs}[1]$ to $D$. 
MATLAB
A call to
MEX-file func:
\[ C, D = \text{func}(A, B) \]
tells MATLAB to pass variables A and B to your MEX-file. C and D are left unassigned.

func.F
subroutine mexFunction(
    nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*), nlhs, nrhs
In the gateway routine:
- Use the mxCreate functions to create the MATLAB arrays for your output arguments. Set plhs(1), (2),... to the pointers to the newly created MATLAB arrays.
- Use the mxGet functions to extract your data from prhs(1), (2),... .
- Call your Fortran subroutine passing the input and output data pointers as function parameters using %val.

MATLAB
On return from MEX-file func:
\[ C, D = \text{func}(A, B) \]
plhs(1) is assigned to C and plhs(2) is assigned to D.

Fortran MEX Cycle
User Messages

To print a string 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.

```matlab
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
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
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:

```plaintext
copyfile(fullfile(matlabroot,'extern','examples','refbook','timestwo.F'),'.','f')
```

Use the `mex` function to build the MEX file.

```plaintext
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.

```plaintext
timestwo(4)
```

```plaintext
ans =
  8
```

See Also

- `mex` | `mexext`

More About

- “Handling Large mxArrays” on page 5-61
- “Upgrade MEX-Files to Use 64-Bit API” on page 4-36
Create Fortran Source MEX File

This example shows how to write a MEX file to call a Fortran subroutine, \texttt{timestwo}, in MATLAB using a MATLAB matrix. \texttt{timestwo} multiplies an \textit{n}-dimensional array, \texttt{x\_input}, by 2, and returns the results in array, \texttt{y\_output}.

\begin{verbatim}
  subroutine timestwo(y_output, x_input)
    real*8 x_input, y_output
    y_output = 2.0 * x_input
    return
  end
\end{verbatim}

Create Source File

Open MATLAB Editor, create a file, and document the MEX file with the following information.

\begin{verbatim}
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, \texttt{fintrf.h}, containing the MATLAB API function declarations.

#include "fintrf.h"

Save the file on your MATLAB path, for example, in \texttt{c:\work}, and name it \texttt{timestwo.F}. The name of your MEX file is \texttt{timestwo}.

Create Gateway Routine

MATLAB uses the gateway routine, \texttt{mexfunction}, as the entry point to a Fortran subroutine. Add the following \texttt{mexFunction} code.

\begin{verbatim}
C Gateway routine
  subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C Declarations
\end{verbatim}
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 mxGetPr
    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 `mxGetPr`.

- Declare local variables for the `mexfunction` arguments.

C     Pointers to input/output mxArrays:
    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
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
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
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
C     Arguments for computational routine:
```
real*8  x_input, y_output

Read Input Array

To point to the input matrix data, use the mxGetPr function.

x_ptr = mxGetPr(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 mxGetPr function to assign the y_ptr argument to plhs(1).

  y_ptr = mxGetPr(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.

```matlab
mex timestwo.F
```

**Test the MEX File**

```matlab
x = 99;
y = timestwo(x)
```

```matlab
y =
    198
```

**See Also**

mexfunction | mwPointer | mwSize | mxCreateDoubleMatrix | mxGetM | mxGetN | mxGetPr | mxIsNumeric

**Related Examples**

• timestwo.F
Set Up Fortran Examples

The “Fortran Matrix Library 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:

```matlab
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:

```matlab
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 “Table of MEX File Source Code Files” on page 5-24.
Pass Scalar Values

Here is a Fortran computational routine that takes a scalar and doubles it.

```fortran
subroutine timestwo(y, x)
  real*8 x, y
  C
  y = 2.0 * x
  return
end
```

To see the same function written in the MEX file format (`timestwo.F`), open the file in MATLAB Editor.

To build this example, at the command prompt type:

```bash
mex timestwo.F
```

This command creates the binary MEX file called `timestwo` with an extension corresponding to the machine type on which you are running. You can now call `timestwo` like a MATLAB function:

```matlab
x = 2;
y = timestwo(x)
```

```
y = 4
```
Pass Strings

Passing strings from MATLAB to a Fortran MEX file is straightforward. The program `revord.F` accepts a string and returns the characters in reverse order. To see the example `revord.F`, open the file in MATLAB Editor.

After checking for the correct number of inputs, the gateway routine `mexFunction` verifies that the input was a row vector string. It then finds the size of the string and places the string into a Fortran character array. For character strings, it is not necessary to copy the data into a Fortran character array using `mxCopyPtrToCharacter`. (`mxCopyPtrToCharacter` is a convenience function for working with MAT-files.)

To build this example, at the command prompt type:

```
mex revord.F
```

Type:

```
x = 'hello world';
y = revord(x)
```

```
y =
dlrow olleh
```
Pass Arrays of Strings

Passing arrays of strings adds a complication to the example “Pass Strings” on page 6-21. Because MATLAB stores elements of a matrix by column instead of by row, the size of the string array must be correctly defined in the Fortran MEX file. The key point is that the row and column sizes as defined in MATLAB must be reversed in the Fortran MEX file. So, when returning to MATLAB, the output matrix must be transposed.

This example places a string array/character matrix into MATLAB as output arguments rather than placing it directly into the workspace.

To build this example, at the command prompt type:

```plaintext
mex passstr.F
```

Type:

```plaintext
passstr
```

to create the 5-by-15 `mystring` matrix. You need to do some further manipulation. The original string matrix is 5-by-15. Because of the way MATLAB reads and orients elements in matrices, the size of the matrix must be defined as `M=15` and `N=5` in the MEX file. After the matrix is put into MATLAB, the matrix must be transposed. The program `passstr.F` illustrates how to pass a character matrix. To see the code `passstr.F`, open the file in MATLAB Editor.

Type:

```plaintext
passstr
```

ans =

MATLAB
The Scientific Computing Environment
by TMW, Inc.
Pass Matrices

In MATLAB, you can pass matrices into and out of MEX files written in Fortran. You can manipulate the MATLAB arrays by using `mxGetPr` and `mxGetPi` to assign pointers to the real and imaginary parts of the data stored in the MATLAB arrays. You can create new MATLAB arrays from within your MEX file by using `mxCreateDoubleMatrix`.

The example `matsq.F` takes a real 2-by-3 matrix and squares each element. To see the source code, open the file in MATLAB Editor.

After performing error checking to ensure that the correct number of inputs and outputs was assigned to the gateway subroutine and to verify the input was in fact a numeric matrix, `matsq.F` creates a matrix. The matrix is copied to a Fortran matrix using `mxCopyPtrToReal8`. Now the computational subroutine can be called, and the return argument is placed into `y_pr`, the pointer to the output, using `mxCopyReal8ToPtr`.

To build this example, at the command prompt type:

```
mex matsq.F
```

For a 2-by-3 real matrix, type:

```matlab
x = [1 2 3; 4 5 6];
y = matsq(x)
```

```
y =
 1   4   9
16  25  36
```

**See Also**

`mxCreateDoubleMatrix` | `mxGetPi` | `mxGetPr`
Pass Integers

The example `matsqint8.F` accepts a matrix of MATLAB type `int8` and squares each element. To see the source code, open the file in MATLAB Editor. Data of type `int8`, a signed 8-bit integer, is equivalent to Fortran type `integer*1`, a signed 1-byte integer. To copy values between MATLAB and Fortran arrays, use the `mxCopyPtrToInteger1` and `mxCopyInteger1ToPtr` functions.

To build this example, at the command prompt type:

```
mex matsqint8.F
```

Type:

```matlab
B = int8([1 2; 3 4; -5 -6]);
y = matsqint8(B)
```

```
y =
     1     4
     9    16
    25    36
```

For information about using other integer data types, consult your Fortran compiler manual.

See Also

`mxCopyInteger1ToPtr` | `mxCopyPtrToInteger1`
Pass Multiple Inputs or Outputs

The plhs and prhs parameters (see “Components of Fortran MEX File” on page 6-2) are vectors containing pointers to the left side (output) variables and right-hand side (input) variables. plhs(1) contains a pointer to the first left side argument, plhs(2) contains a pointer to the second left side argument, and so on. Likewise, prhs(1) contains a pointer to the first right-hand side argument, prhs(2) points to the second, and so on.

The example xtimesy.F multiplies an input scalar times an input scalar or matrix. To see the source code, open the file in MATLAB Editor.

As this example shows, creating MEX file gateways that handle multiple inputs and outputs is straightforward. Keep track of which indices of the vectors prhs and plhs correspond to which input and output arguments of your function. In this example, the input variable x corresponds to prhs(1) and the input variable y to prhs(2).

To build this example, at the command prompt type:

mex xtimesy.F

For an input scalar x and a real 3-by-3 matrix, type:

```matlab
x = 3;
y = ones(3);
z = xtimesy(x, y)
```

```
z =
  3  3  3
  3  3  3
  3  3  3
```
Handle Complex Data

MATLAB stores complex double-precision data as two vectors of numbers—one vector contains the real data and the other contains the imaginary data. The functions `mxCopyPtrToComplex16` and `mxCopyComplex16ToPtr` copy MATLAB data to a native `complex*16` Fortran array.

The example `convec.F` takes two complex vectors (of length 3) and convolves them. To see the source code, open the file in MATLAB Editor.

To build this example, at the command prompt type:

```
mex convec.F
```

Enter the following at the command prompt:

```
x = [3 - 1i, 4 + 2i, 7 - 3i];
y = [8 - 6i, 12 + 16i, 40 - 42i];
```

Type:

```
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
```

which agrees with the results the built-in MATLAB function `conv.m` produces.
Dynamically Allocate Memory

To allocate memory dynamically in a Fortran MEX file, use %val. (See “Using the Fortran %val Construct” on page 6-6.) 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:

```matlab
mex dblmat.F compute.F
```

For the 2-by-3 matrix, type:

```matlab
x = [1 2 3; 4 5 6];
y = dblmat(x)
```

```matlab
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.
Handle Sparse Matrices

MATLAB provides a set of functions that allow you to create and manipulate sparse matrices. There are special parameters associated with sparse matrices, namely ir, jc, and nzmax. For information on how to use these parameters and how MATLAB stores sparse matrices in general, see “Sparse Matrices” on page 4-11.

**Note:** Sparse array indexing is zero-based, not one-based.

The `fulltosparse.F` example illustrates how to populate a sparse matrix. To see the source code, open the file in MATLAB Editor. `loadsparse.F` is the subroutine `fulltosparse` calls to fill the `mxArray` with the sparse data. (Open the file in MATLAB Editor.)

To build this example, at the command prompt type:

```
mex fulltosparse.F loadsparse.F
```

At the command prompt, type:

```
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
```

MATLAB creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix:

```
spar = fulltosparse(full)
```

```
spar =
(1,1)  1
(2,2)  1
(3,3)  1
(4,4)  1
(5,5)  1
```
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 `eig`, do not work correctly with data that is not double precision.

To build this example, at the command prompt type:

```
mex 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:

```matlab
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 Source MEX-Files

In this section...

“Notes on Debugging” on page 6-31
“Debugging on Microsoft Windows Platforms” on page 6-31
“Debugging on Linux Platforms” on page 6-31

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 “Troubleshoot MEX Files”.

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. Refer to the “Creating C/C++ Language MEX Files” topic “Debugging on Microsoft Windows Platforms” on page 5-52 for instructions on using this debugger.

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:
At the Linux prompt, start the gdb debugger using the `matlab -D` option:

```bash
linux> matlab -Dgdb
```

Start MATLAB without the Java Virtual Machine (JVM) by using the `-nojvm` startup flag:

```bash
<gdb> run -nojvm
```

In MATLAB, enable debugging with the `dbmex` function and run your binary MEX file:

```matlab
>> dbmex on
>> y = timestwo(4)
```

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:

```bash
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.

```bash
<gdb> break mexfunction_
<gdb> continue
```

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`:

```bash
<gdb> continue
```
After stopping at the last breakpoint, type:

```gdb> continue```

timestwo finishes and MATLAB displays:

```y =
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```

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.

**More About**

- “Fortran Source Files”
Handling Large `mxArray`s

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Using the 64-Bit API” on page 6-34</td>
</tr>
<tr>
<td>“Building the Binary MEX File” on page 6-36</td>
</tr>
<tr>
<td>“Caution Using Negative Values” on page 6-36</td>
</tr>
<tr>
<td>“Building Cross-Platform Applications” on page 6-36</td>
</tr>
</tbody>
</table>

Binary MEX-files built on 64-bit platforms can handle 64-bit `mxArray`s. 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”. Memory management within source MEX-files can have special considerations, as described in “Memory Management” on page 5-65.

**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.

<table>
<thead>
<tr>
<th><code>mxArray</code> Functions Using <code>mwSize/mwIndex</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mxCalcSingleSubscript</code></td>
</tr>
<tr>
<td><code>mxAlloc</code></td>
</tr>
<tr>
<td><code>mxCopyCharacterToPtr</code>¹</td>
</tr>
<tr>
<td>Function Name</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>mxCopyComplex16ToPtr</td>
</tr>
<tr>
<td>mxCopyComplex8ToPtr</td>
</tr>
<tr>
<td>mxCopyInteger1ToPtr</td>
</tr>
<tr>
<td>mxCopyInteger2ToPtr</td>
</tr>
<tr>
<td>mxCopyInteger4ToPtr</td>
</tr>
<tr>
<td>mxCopyPtrToCharacter</td>
</tr>
<tr>
<td>mxCopyPtrToComplex16</td>
</tr>
<tr>
<td>mxCopyPtrToComplex8</td>
</tr>
<tr>
<td>mxCopyPtrToInteger1</td>
</tr>
<tr>
<td>mxCopyPtrToInteger2</td>
</tr>
<tr>
<td>mxCopyPtrToInteger4</td>
</tr>
<tr>
<td>mxCopyPtrToPtrArray</td>
</tr>
<tr>
<td>mxCopyPtrToReal4</td>
</tr>
<tr>
<td>mxCopyPtrToReal8</td>
</tr>
<tr>
<td>mxCreateCharArray</td>
</tr>
<tr>
<td>mxCreateCellArray</td>
</tr>
<tr>
<td>mxCreateCharArrayFromStrings</td>
</tr>
<tr>
<td>mxCreateDoubleMatrix</td>
</tr>
<tr>
<td>mxCreateLogicalArray</td>
</tr>
<tr>
<td>mxCreateLogicalMatrix</td>
</tr>
<tr>
<td>mxCreateNumericArray</td>
</tr>
<tr>
<td>mxCreateNumericMatrix</td>
</tr>
<tr>
<td>mxCreateSparse</td>
</tr>
</tbody>
</table>

1 Fortran function only.
Fortran MEX-Files

\(^2\)C function only.

Functions in this API use the `mwIndex`, `mwSize`, and `mwPointer` preprocessor macros.

**Building the Binary MEX File**

Use the `mex` build script option `-largeArrayDims` with the 64-bit API.

**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- 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 file 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 file that is not in plhs. MATLAB also frees any memory that was allocated in the MEX file using the mxCalloc, mxMalloc, or mxRealloc 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-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. For more information on memory management techniques, see the sections “Memory Management” on page 5-65 in Creating C/C++ Language MEX Files and “Memory Management Issues” on page 4-67.
MATLAB Supports Fortran 77


For example, the length of the following statement is less 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.
Calling MATLAB Engine from C/C++ and Fortran Programs

- “Introducing MATLAB Engine API for C/C++ and Fortran” on page 7-2
- “Callbacks in Applications” on page 7-4
- “Call MATLAB Functions from C/C++ Applications” on page 7-5
- “Call MATLAB Functions from Fortran Applications” on page 7-7
- “Attach to Existing MATLAB Sessions” on page 7-9
- “Build Windows Engine Application” on page 7-11
- “Run Windows Engine Application” on page 7-13
- “Set Run-Time Library Path on Windows Systems” on page 7-14
- “Register MATLAB as a COM Server” on page 7-16
- “Build Linux Engine Application” on page 7-17
- “Run Linux Engine Application” on page 7-18
- “Set Run-Time Library Path on Mac and Linux Systems” on page 7-19
- “Build Engine Applications with IDE” on page 7-21
- “Can't Start MATLAB Engine” on page 7-24
- “Debug MATLAB Functions Used in Engine Applications” on page 7-25
- “Multithreaded Applications” on page 7-26
- “User Input Not Supported” on page 7-27
- “Getting Started” on page 7-28
Introducing MATLAB Engine API for C/C++ and Fortran

The MATLAB C/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/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.

More About
- “MATLAB COM Integration” on page 11-2
- “MATLAB Engine for Python”
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:

```matlab
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/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 the Microsoft Windows version of this program, see engwindemo.c.

To see engdemo.c, open this file in MATLAB Editor.

To see the Windows version engwindemo.c, open this file.

There is a C++ version of engdemo in the matlabroot\extern\examples\eng_mat folder. To see engdemo.cpp, open this file.

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.

**Related Examples**
- “Build Windows Engine Application” on page 7-11
- “Build Linux Engine Application” on page 7-17

**More About**
- “Build Engine Applications with IDE” on page 7-21
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. You should only call engOpen 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 Mac 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
```

Click OK. This starts MATLAB.

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 does not start another MATLAB session, but rather uses the MATLAB session that is already open.

**More About**

- “Launch MATLAB as Automation Server in Desktop Mode” on page 13-16
- “MATLAB COM Integration” on page 11-2
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')

Build the application.

mex -v -client engine engwindemo.c

Build Standalone Application with MinGW Compiler

If you have installed one of the professional editions of the Microsoft Visual Studio compiler and want to build with the MinGW compiler, use the following command to override the default Microsoft compiler.

mex -v -client engine -f matlabroot\bin\win64\mexopts\mingw64.xml engwindemo.c

If the only compiler on your system is the MinGW compiler, your build command is:

mex -v -client engine engwindemo.c

See Also
mex

Related Examples
• “Call MATLAB Functions from C/C++ Applications” on page 7-5
• “Run Windows Engine Application” on page 7-13
More About

- “Register MATLAB as a COM Server” on page 7-16
Run Windows Engine Application

This example shows how to run the C example, engwindemo.c, from the Windows system prompt. You need to know 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.

Click **Ok** in the MATLAB `whos` window to close the application.

**See Also**

`computer` | `matlabroot`

**Related Examples**

- "Build Windows Engine Application" on page 7-11
- "Call MATLAB Functions from C/C++ Applications" on page 7-5

**More About**

- "Set Run-Time Library Path on Windows Systems" on page 7-14
Set Run-Time Library Path on Windows Systems

In this section...

“Change Path Each Time You Run the Application” on page 7-14
“Permanently Change Path” on page 7-14
“Troubleshooting” on page 7-15

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 string 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 string returned from the fullfile command. Set the path every time you open the Windows Command Processor.

set PATH=res;%PATH%

You must 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.
• Click **Ok** to close the dialog boxes, 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. Do this 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/C++ Applications” on page 7-5
• “Run Linux Engine Application” on page 7-18
Run Linux Engine Application

This example shows how to run the C example, engdemo.c, from the Linux system prompt. You need to know the value of `matlabroot`, the folder where MATLAB is installed.

Set the run-time library path.

```bash
setenv LD_LIBRARY_PATH matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64
```

This command replaces the value, if any, in `LD_LIBRARY_PATH`.

Set the path.

```bash
setenv PATH matlabroot/bin:$PATH
```

Make sure that you include the `:` path terminator character.

Run the example. The `engdemo` application must be on your system path.

```bash
./engdemo
```

MATLAB starts and displays a figure.

Press Return to close the figure.

Create variable `X`, for example:

```bash
X = 'hello'
```

MATLAB displays the results and closes.

**Related Examples**

- “Build Linux Engine Application” on page 7-17
- “Call MATLAB Functions from C/C++ Applications” on page 7-5

**More About**

- “Set Run-Time Library Path on Mac and Linux Systems” on page 7-19
Set Run-Time Library Path on Mac and Linux Systems

At run time, tell the operating system where the API shared libraries reside by setting an environment variable. Set the path every time you run MATLAB. Alternatively, place the commands in a MATLAB startup script.

The shell command you use and the value you provide to set the run-time library path 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. In the `pathspec` string, replace `matlabroot` with the string returned by the `matlabroot` command.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>envvar</th>
<th>pathspec</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Apple Mac</td>
<td>DYLD_LIBRARY_PATH</td>
<td><code>matlabroot/bin/mac164:matlabroot/sys/os/mac164</code></td>
</tr>
<tr>
<td>64-bit Linux</td>
<td>LD_LIBRARY_PATH</td>
<td><code>matlabroot/bin/glnxa64:matlabroot/sys/os/glnxa64</code></td>
</tr>
</tbody>
</table>

These commands replace the existing value in `envvar`. If `envvar` is already defined, prepend the `pathspec` 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.

C Shell

Set the library path using the following command, replacing the terms `envvar` and `pathspec` with the appropriate values from the table.

```
setenv envvar pathspec
```

For example, for MATLAB R2015a on a Mac system:

```
setenv DYLD_LIBRARY_PATH /Applications/MATLAB_R2015a.app/bin/maci64:/Applications/MATLAB...
```

You can place these commands in a startup script, such as `~/.cshrc`.  

7-19
### Bourne Shell

Set the library path using the following command, replacing the terms `envvar` and `pathspec` with the appropriate values from the table.

```
envvar = pathspec:envvar export envvar
```

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`
Build Engine Applications with IDE

In this section...

“Configuring the IDE” on page 7-21
“Engine Include Files” on page 7-21
“Engine Libraries” on page 7-21
“Library Files Required by libeng” on page 7-22

Configuring the IDE

If your integrated development environment (IDE) has a MATLAB-supported compiler, you can use the IDE to build engine applications. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.

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, Mac, 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.

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, as shown in the following table.
Library File Names by Operating System

<table>
<thead>
<tr>
<th>Windows⁴</th>
<th>Linux</th>
<th>Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>libeng.lib</td>
<td>libeng.so</td>
<td>libeng.dylib</td>
</tr>
<tr>
<td>libmx.lib</td>
<td>libmx.so</td>
<td>libmx.dylib</td>
</tr>
</tbody>
</table>

⁴ These files are located in the folder `fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft')`.

Add these library names to your IDE configuration. Set the library path to the value returned by the following MATLAB command:

```
fullfile(matlabroot,'bin',computer('arch'))
```

Refer to your IDE product documentation for instructions.

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**

<table>
<thead>
<tr>
<th>Windows</th>
<th>Linux</th>
<th>Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>See the following instructions for Dependency Walker</td>
<td><code>ldd -d libeng.so</code></td>
<td><code>otool -L libeng.dylib</code></td>
</tr>
</tbody>
</table>

On Windows systems, to find library dependencies, use the third-party product Dependency Walker. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module 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. Download the Dependency Walker utility from the website [http://www.dependencywalker.com/](http://www.dependencywalker.com/). See [http://www.mathworks.com/matlabcentral/answers/92362-how-do-i-determine-which-libraries-my-mex-file-or-stand-alone-application-requires](http://www.mathworks.com/matlabcentral/answers/92362-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.

**Related Examples**
- “Build Windows Engine Application” on page 7-11
- “Build Linux Engine Application” on page 7-17

**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

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.

Related Examples
• “Register MATLAB as a COM Server” on page 7-16
Debug MATLAB Functions Used in Engine Applications

When creating MATLAB functions for use in engine applications, it is good practice to debug the functions in MATLAB before calling them through the engine library functions.

Although you cannot use the MATLAB Editor/Debugger from an engine application, you can use the MATLAB workspace to examine variables passed to MATLAB. For example, you have the following MATLAB function:

```matlab
function y = myfcn(x)
y = x+2;
end
```

Your engine application calls `myfcn` with your variable `mycmxarray`, as shown in the following code:

```matlab
engPutVariable(ep, "aVar", mycmxarray);
engEvalString(ep, "result = myfcn(aVar)" unfortunate typo corrected); mycmxarrayResult = engGetVariable(ep,"result");
```

If you do not get the expected result, you can examine two possibilities: if the input, `mycmxarray`, is incorrect, or if the MATLAB function is incorrect.

To examine the input to `myfcn`, first modify the function to save the MATLAB workspace to the file `debugmyfcn.mat`.

```matlab
function y = myfcn(x)
save debugmyfcn.mat
y = x+2;
end
```

Execute your engine application, then start MATLAB and load `debugmyfcn.mat`.

```matlab
load debugmyfcn.mat
whos x
```

Variable `x` contains the value from `mycmxarray`. If `x` is not what you expect, debug your engine code. If `x` is correct, debug the MATLAB function. To debug `myfcn`, open the function in the MATLAB Editor/Debugger, and then call the function from the MATLAB command line:

```matlab
myfcn(x)
```
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 or disrupt program flow are not supported. Examples of such functions include debugging commands, `input`, `keyboard`, and `pause`.
Getting Started

To build an engine application, you need:

- The ability to write C/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 the Supported and Compatible Compilers website.
- “C/C++ Matrix Library API” or “Fortran Matrix Library API” functions.
- 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 7-21.

To run the application:

- “Set Run-Time Library Path on Windows Systems” on page 7-14
- “Register MATLAB as a COM Server” on page 7-16
- “Set Run-Time Library Path on Mac and Linux Systems” on page 7-19
MATLAB Engine for Python Topics

- “Get Started with MATLAB Engine for Python” on page 8-2
- “Install MATLAB Engine for Python” on page 8-5
- “Start and Stop MATLAB Engine for Python” on page 8-8
- “Connect Python to Running MATLAB Session” on page 8-10
- “Call MATLAB Functions from Python” on page 8-13
- “Call MATLAB Functions Asynchronously from Python” on page 8-15
- “Call User Script and Function from Python” on page 8-16
- “Redirect Standard Output and Error to Python” on page 8-18
- “Use MATLAB Handle Objects in Python” on page 8-19
- “Use MATLAB Engine Workspace in Python” on page 8-21
- “Pass Data to MATLAB from Python” on page 8-22
- “Handle Data Returned from MATLAB to Python” on page 8-24
- “MATLAB Arrays as Python Variables” on page 8-27
- “Use MATLAB Arrays in Python” on page 8-32
- “Sort and Plot MATLAB Data from Python” on page 8-34
- “Get Help for MATLAB Functions from Python” on page 8-38
- “Default Numeric Types in MATLAB and Python” on page 8-40
- “System Requirements for MATLAB Engine for Python” on page 8-42
- “Limitations to MATLAB Engine for Python” on page 8-44
- “Troubleshoot MATLAB Errors in Python” on page 8-45
Get Started with MATLAB Engine for Python

The MATLAB Engine 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 for Python” on page 8-5
- “Start and Stop MATLAB Engine for Python” on page 8-8

The `matlab` package contains the following:

- The MATLAB Engine for Python
- A set of MATLAB array classes in Python (see “MATLAB Arrays as Python Variables” on page 8-27)

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.

<table>
<thead>
<tr>
<th>Package</th>
<th>Function or Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matlab.engine</code></td>
<td><code>start_matlab()</code></td>
<td>Python function to create a <code>MatlabEngine</code> object, and attach it to a new MATLAB process</td>
</tr>
<tr>
<td><code>matlab.engine</code></td>
<td><code>MatlabEngine</code></td>
<td>Python class to provide methods for calling MATLAB functions</td>
</tr>
<tr>
<td><code>matlab.engine</code></td>
<td><code>FutureResult</code></td>
<td>Python class to hold results from a MATLAB process</td>
</tr>
<tr>
<td>Package</td>
<td>Function or Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>matlab</td>
<td>double</td>
<td>Python class to hold array of MATLAB type double</td>
</tr>
<tr>
<td>matlab</td>
<td>single</td>
<td>Python class to hold array of MATLAB type single</td>
</tr>
<tr>
<td>matlab</td>
<td>int8</td>
<td>Python class to hold array of MATLAB type int8</td>
</tr>
<tr>
<td>matlab</td>
<td>int16</td>
<td>Python class to hold array of MATLAB type int16</td>
</tr>
<tr>
<td>matlab</td>
<td>int32</td>
<td>Python class to hold array of MATLAB type int32</td>
</tr>
<tr>
<td>matlab</td>
<td>int64</td>
<td>Python class to hold array of MATLAB type int64</td>
</tr>
<tr>
<td>matlab</td>
<td>uint8</td>
<td>Python class to hold array of MATLAB type uint8</td>
</tr>
<tr>
<td>matlab</td>
<td>uint16</td>
<td>Python class to hold array of MATLAB type uint16</td>
</tr>
<tr>
<td>matlab</td>
<td>uint32</td>
<td>Python class to hold array of MATLAB type uint32</td>
</tr>
<tr>
<td>matlab</td>
<td>uint64</td>
<td>Python class to hold array of MATLAB type uint64</td>
</tr>
<tr>
<td>Package</td>
<td>Function or Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>matlab</td>
<td>logical</td>
<td>Python class to hold array of MATLAB type <code>logical</code></td>
</tr>
<tr>
<td>matlab</td>
<td>object</td>
<td>Python class to hold a handle to a MATLAB object</td>
</tr>
</tbody>
</table>
Install MATLAB Engine for Python

To start the MATLAB engine within a Python session, you first must install the engine as a Python package. The engine supports Python versions 2.7, 3.3, and 3.4, and MATLAB versions R2014b and later. You can install and run the engine on any platform that MATLAB supports. You can install the engine in a nondefault folder. When you have multiple versions of Python on your system, or you install the engine in a nondefault folder, use an environment variable to specify run-time paths.

Verify Python and MATLAB Installations

First, verify that your system has the correct versions of Python and MATLAB. Then, find the path to the MATLAB folder. You need the path to the MATLAB folder to install the MATLAB Engine for Python.

1. Check that Python is installed on your system and that you can run Python at the operating system prompt.
   - To install Python 2.7, 3.3, or 3.4, see “Install Supported Python Implementation” on page 1-3.
2. Add the folder that contains the Python interpreter to your path, if it is not already there.
3. Find the path to the MATLAB folder. Start MATLAB and type `matlabroot` in the command window. Copy the path returned by `matlabroot`.

Install Engine

To install the MATLAB Engine for Python, execute the following commands where `matlabroot` is the path to the MATLAB folder.

Windows system:

```
cd "matlabroot\extern\engines\python"
python setup.py install
```

Mac or Linux system:

```
cd "matlabroot/extern/engines/python"
python setup.py install
```
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. For more information, see Installing Python Modules.

**Build Engine in Nondefault Folder**

To build the MATLAB Engine for Python in a nondefault folder, execute the following commands at the operating system prompt. `matlabroot` is the path to the MATLAB folder. Build the engine in folder `builddir`. You can use these commands when you do not have write permission to build the engine in the MATLAB folder.

```
cd "matlabroot\extern\engines\python"
python setup.py build --build-base="builddir" install
```

**Install Engine in Nondefault Folder**

To install the MATLAB Engine for Python in a nondefault folder, execute the following commands at the operating system prompt. `matlabroot` is the path to the MATLAB folder. Install the engine in folder `installdir`. You can use these commands when you do not have write permission to install the engine in the default Python folder.

```
cd "matlabroot\extern\engines\python"
python setup.py install --prefix="installdir"
```

You must include `installdir` in the search path for Python packages. Add `installdir` to the PYTHONPATH environment variable.

You also can combine the commands to build and install the engine in nondefault folders.

```
cd "matlabroot\extern\engines\python"
python setup.py build --build-base="builddir" install --prefix="installdir"
```

**Install Engine in Your Home Folder**

To install the MATLAB Engine for Python for your use only, use the --user option to install to 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.
Set Run-Time Paths To Python Code

You can install the engine, or Python code of your own, in nondefault folders. Add paths to nondefault folders to the PYTHONPATH environment variable. Python searches PYTHONPATH for modules to import.

To specify the path to the Python library, or to another library of your own, add the library path to the appropriate environment variable. This table lists the appropriate environment variable for each system.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>PATH</td>
</tr>
<tr>
<td>Mac</td>
<td>DYLD_LIBRARY_PATH</td>
</tr>
<tr>
<td>Linux</td>
<td>LD_LIBRARY_PATH</td>
</tr>
</tbody>
</table>

Modify your library path to link to:

- A specific version of the engine shared library, when you have installed multiple versions of the engine
- A specific version of the Python shared library, if your system has multiple versions of Python
- A shared library of your own code

More About
- “System Requirements for MATLAB Engine for Python” on page 8-42
Start and Stop MATLAB Engine for Python

In this section...

“Start MATLAB Engine for Python” on page 8-8
“Run Multiple Engines” on page 8-8
“Stop MATLAB Engine” on page 8-8
“Start Engine with Startup Options” on page 8-9

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 allows you to pass data and call functions executed by MATLAB.

```python
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.

```python
eng1 = matlab.engine.start_matlab()
eng2 = matlab.engine.start_matlab()
```

Stop MATLAB Engine

Call either the `exit` or the `quit` function.

```python
eng.quit()
```

If you exit Python with an engine still running, 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.

```python
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.

```python
eng = matlab.engine.start_matlab("-desktop -r 'format short'")
```

You also can start the desktop after you start the engine.

```python
import matlab.engine
eng = matlab.engine.start_matlab()
eng.desktop(nargout=0)
```

More About

- “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`.

```python
matlab.engine.shareEngine
```

Start Python at the operating system prompt. Call `matlab.engine.connect_matlab` from Python to connect to the shared MATLAB session. You can call any MATLAB function from Python.

```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.

```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.

```python
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 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”.

```python
matlab.engine.shareEngine('MATLABEngine2')
```

From Python find all shared MATLAB sessions.

```python
import matlab.engine
matlab.engine.find_matlab()

('MATLAB_13232', 'MATLABEngine2')
```

Call `matlab.engine.connect_matlab` from Python to connect to the shared MATLAB sessions.

```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.

```bash
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

- “Startup Options”
- “Commonly Used Startup Options”
Call MATLAB Functions from Python

In this section...

| “Return Output Argument from MATLAB Function” on page 8-13 |
| “Return Multiple Output Arguments from MATLAB Function” on page 8-13 |
| “Return No Output Arguments from MATLAB Function” on page 8-13 |
| “Stop Execution of Function” on page 8-14 |

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.

```python
import matlab.engine
ing = matlab.engine.start_matlab()
print(eng.isprime(37))
```

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`.

```python
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.

8-13
Open the MATLAB Help browser from Python.

```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.MatlabEngine`
- `matlab.engine.FutureResult`

**Related Examples**

- “Call MATLAB Functions Asynchronously from Python” on page 8-15
- “Call User Script and Function from Python” on page 8-16
- “Use MATLAB Arrays in Python” on page 8-32
- “Sort and Plot MATLAB Data from Python” on page 8-34
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 `async` argument to call a MATLAB function asynchronously.

```python
import matlab.engine
eng = matlab.engine.start_matlab()
future = eng.sqrt(4.0, async=True)
ret = future.result()
print(ret)
2.0
```

Use the `done` method to check if an asynchronous call finished.

```python
tf = future.done()
print(tf)
True
```

To stop execution of the function before it finishes, call `future.cancel()`.

**See Also**
matlab.engine.MatlabEngine | matlab.engine.FutureResult

**Related Examples**
- “Call MATLAB Functions from Python” on page 8-13
- “Call User Script and Function from Python” on page 8-16
Call User Script and Function 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.

```matlab
b = 5;
h = 3;
a = 0.5*(b.* h)
```

After you save the file, start Python and call the script.

```python
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. Open the MATLAB editor to edit the file.

```matlab
function a = triarea(b,h)
a = 0.5*(b.* h);
```

Call the new triarea function from the engine.

```python
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.MatlabEngine | matlab.engine.FutureResult

Related Examples
• “Call MATLAB Functions from Python” on page 8-13
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. Specify `stdout` and `stderr` to capture a warning message from `dec2hex`.

```python
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 (2^52). Results may be unpredictable.
```

In Python 3.3 or 3.4, use the `io` module to create `StringIO` objects.

```python
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 2^52. 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.MatlabEngine`
- `matlab.engine.FutureResult`

Related Examples

- “Call MATLAB Functions from Python” on page 8-13
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.

```matlab
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.

```python
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.

```python
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.

```python
eng.setHeight(tr,8.0,nargout=0)
a = eng.area(tr)
print(a)

20.0
```

**See Also**

`matlab.engine.MatlabEngine` | `matlab.engine.FutureResult`

**Related Examples**

- “Call MATLAB Functions from Python” on page 8-13
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.

```python
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 new 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.MatlabEngine | matlab.engine.FutureResult

Related Examples
- “Call MATLAB Functions from Python” on page 8-13
- “Sort and Plot MATLAB Data from Python” on page 8-34
Pass Data to MATLAB from Python

In this section...

- “Python Type to MATLAB Scalar Type Mapping” on page 8-22
- “Python Container to MATLAB Array Type Mapping” on page 8-22
- “Unsupported Python Types” on page 8-23

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.

<table>
<thead>
<tr>
<th>Python Input Argument Type — Scalar Values Only</th>
<th>Resulting MATLAB Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>complex</td>
<td>Complex double</td>
</tr>
<tr>
<td>int</td>
<td>int64</td>
</tr>
<tr>
<td>long (Python 2.7 only)</td>
<td>int64</td>
</tr>
<tr>
<td>float(nan)</td>
<td>NaN</td>
</tr>
<tr>
<td>float(inf)</td>
<td>Inf</td>
</tr>
<tr>
<td>bool</td>
<td>logical</td>
</tr>
<tr>
<td>str</td>
<td>char</td>
</tr>
<tr>
<td>unicode (Python 2.7 only)</td>
<td>char</td>
</tr>
<tr>
<td>dict</td>
<td>Structure if all keys are strings not supported otherwise</td>
</tr>
</tbody>
</table>

Python Container to MATLAB Array Type Mapping

<table>
<thead>
<tr>
<th>Python Input Argument Type — Container</th>
<th>Resulting MATLAB Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>matlab numeric array object (see “MATLAB Arrays as Python Variables” on page 8-27)</td>
<td>Numeric array</td>
</tr>
</tbody>
</table>

8-22
Pass Data to MATLAB from Python

<table>
<thead>
<tr>
<th>Python Input Argument Type — Container</th>
<th>Resulting MATLAB Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytearray</td>
<td>uint8 array</td>
</tr>
<tr>
<td>bytes (Python 3.3 and 3.4)</td>
<td>uint8 array</td>
</tr>
<tr>
<td>bytes (Python 2.7)</td>
<td>char array</td>
</tr>
<tr>
<td>list</td>
<td>Cell array</td>
</tr>
<tr>
<td>set</td>
<td>Cell array</td>
</tr>
<tr>
<td>tuple</td>
<td>Cell array</td>
</tr>
</tbody>
</table>

**Unsupported Python Types**

The following Python types are not supported by the MATLAB Engine for Python:

- `array.array` (use MATLAB numeric array objects instead; see “MATLAB Arrays as Python Variables” on page 8-27)
- None
- `module.type` object
Handle Data Returned from MATLAB to Python

<table>
<thead>
<tr>
<th>MATLAB Output Argument Type — Scalar Values Only</th>
<th>Resulting Python Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>single</td>
<td>float</td>
</tr>
<tr>
<td>Complex (any numeric type)</td>
<td>complex</td>
</tr>
<tr>
<td>int8</td>
<td>int</td>
</tr>
<tr>
<td>uint8</td>
<td>int</td>
</tr>
<tr>
<td>int16</td>
<td>int</td>
</tr>
<tr>
<td>uint16</td>
<td>int</td>
</tr>
<tr>
<td>int32</td>
<td>int</td>
</tr>
<tr>
<td>uint32</td>
<td>int (Python 3.3 and 3.4)</td>
</tr>
<tr>
<td>int64</td>
<td>int (Python 3.3 and 3.4)</td>
</tr>
<tr>
<td>uint64</td>
<td>int (Python 3.3 and 3.4)</td>
</tr>
<tr>
<td>NaN</td>
<td>float(nan)</td>
</tr>
<tr>
<td>Inf</td>
<td>float(inf)</td>
</tr>
<tr>
<td>logical</td>
<td>bool</td>
</tr>
<tr>
<td>char returned to Python 3.3 or 3.4</td>
<td>str</td>
</tr>
</tbody>
</table>
Handle Data Returned from MATLAB to Python

<table>
<thead>
<tr>
<th>MATLAB Output Argument Type — Scalar Values Only</th>
<th>Resulting Python Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char returned to Python 2.7</td>
<td>str (when MATLAB char value is less than or equal to \text{intmax('uint8')}))</td>
</tr>
<tr>
<td></td>
<td>unicode (when MATLAB char value is greater than \text{intmax('uint8')}))</td>
</tr>
<tr>
<td>Structure</td>
<td>dict</td>
</tr>
<tr>
<td>MATLAB handle object (such as the containers.Map type)</td>
<td>matlab.object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MATLAB Array Type to Python Type Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB Output Argument Type — Array</td>
</tr>
<tr>
<td>Numeric array</td>
</tr>
<tr>
<td>char array (1-by-N, N-by-1) returned to Python 3.3 or 3.4</td>
</tr>
<tr>
<td>char array (1-by-N, N-by-1) returned to Python 2.7</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Row or column cell array</td>
</tr>
</tbody>
</table>

Unsupported MATLAB Types

The following MATLAB data types are not supported by the MATLAB Engine for Python:

- Categorical array
- char array (M-by-N)
- Cell array (M-by-N)
- Function handle
- Sparse array
• Structure array
• Table
• MATLAB value objects (for a discussion of handle and value classes see “Comparison of Handle and Value Classes”)
• Non-MATLAB objects (such as Java objects)
MATLAB Arrays as Python Variables

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 pass MATLAB arrays as input arguments to functions called with the MATLAB Engine 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.

<table>
<thead>
<tr>
<th><code>matlab</code> Class</th>
<th>Constructor Call in Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matlab.double</code></td>
<td><code>matlab.double(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab Class</td>
<td>Constructor Call in Python</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>matlab.single</td>
<td><code>matlab.single(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.int8</td>
<td><code>matlab.int8(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.int16</td>
<td><code>matlab.int16(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.int32</td>
<td><code>matlab.int32(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.int64&lt;sup&gt;a&lt;/sup&gt;</td>
<td><code>matlab.int64(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.uint8</td>
<td><code>matlab.uint8(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.uint16</td>
<td><code>matlab.uint16(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.uint32</td>
<td><code>matlab.uint32(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.uint64&lt;sup&gt;b&lt;/sup&gt;</td>
<td><code>matlab.uint64(initializer=None, size=None, is_complex=False)</code></td>
</tr>
<tr>
<td>matlab.logical</td>
<td><code>matlab.logical(initializer=None, size=None)</code></td>
</tr>
<tr>
<td>matlab.object</td>
<td>No constructor. When a function returns a handle to a MATLAB object, the engine returns a <code>matlab.object</code> to Python.</td>
</tr>
</tbody>
</table>

---

<sup>a</sup> In Python 2.7 on Windows, `matlab.int64` is converted to `int32` in MATLAB. Also, MATLAB cannot return an `int64` array to Python.  
<sup>b</sup> In Python 2.7 on Windows, `matlab.uint64` is converted to `uint32` in MATLAB. Also, MATLAB cannot return a `uint64` array to Python.  
<sup>c</sup> 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.

```python
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.

<table>
<thead>
<tr>
<th>Attribute or Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Size of array returned as a tuple</td>
</tr>
<tr>
<td>reshape(size)</td>
<td>Reshape array as specified by sequence size</td>
</tr>
</tbody>
</table>

**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.

```python
import matlab.engine
A = matlab.double([[1,2,3,4,5], [6,7,8,9,10]])
print(A)
```

```plaintext
[[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.

```python
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.

```python
import matlab.engine
A = matlab.int8([1,2,3,4,5])
p = print(A[0])
```

```plaintext
1
```
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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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 below.

```python
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.

```python
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]]
```
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 \texttt{sqrt} function.

The \texttt{matlab} package provides constructors to create MATLAB arrays in Python. The MATLAB Engine 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 \texttt{sqrt} function on the array.

```python
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 \texttt{b}, which is a 1-by-5 \texttt{matlab.double} array.

Create a multidimensional array. The \texttt{magic} function returns a 2-D \texttt{matlab.double} array to Python. Use a \texttt{for} loop to print each row on a separate line. (Press \texttt{Enter} again when you see the \ldots prompt to close the loop and print.)

```python
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 \texttt{tril} function to get the lower triangular portion of \texttt{a}. Print each row on a separate line.

```python
b = eng.tril(a)
for x in b: print(x)
...  
```
Use MATLAB Arrays in Python

Related Examples

• “Call MATLAB Functions from Python” on page 8-13

More About

• “MATLAB Arrays as Python Variables” on page 8-27
Sort and Plot MATLAB Data from Python

This example shows how to sort data about patients into lists of smokers and non-smokers 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.

```python
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.

```python
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`.

```python
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 non-smokers. In patients.dat, the column `Smoker` indicated a smoker with logical 1 (true), and a non-smoker 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 nonsmokers. 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))
89.917647059

Display the mean blood pressure for the nonsmokers.

print(eng.mean(nsp))
79.37878788
Plot blood pressure readings for the smokers and nonsmokers. Call the MATLAB `linspace` function to define two x-axes for plotting. You can plot the 34 smokers and 66 nonsmokers on the same scatter plot.

```matlab
sdx = eng.linspace(1.0,34.0,34)
nsdx = eng.linspace(1.0,34.0,66)
```

Show the axes boundaries with the `box` function.

```matlab
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’s properties in Python. You can pass MATLAB objects as input arguments to other MATLAB functions.

```matlab
eng.scatter(sdx,sp,10,'blue')
```

```matlab
<matlab.object object at 0x22d1510>
```

In the rest of this example, assign the output argument of MATLAB functions to `h` as a placeholder.

```matlab
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 to show the average blood pressure readings for smokers and nonsmokers.

```matlab
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")
```
Get Help for MATLAB Functions from Python

In this section...

“How to Find MATLAB Help” on page 8-38
“Open MATLAB Help Browser from Python” on page 8-38
“Display MATLAB Help at Python Prompt” on page 8-39

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 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`.)

```python
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**

Call the MATLAB help function to display help text for a function at the Python prompt. 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:

   \[ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, dt. \]

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. This differs from Python, which 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:

```matlab
x = 4;
y = 4.0;
```

Both `x` and `y` are of data type `double`. Now consider the same assignments in Python:

```python
x = 4
y = 4.0
```

`x` and `y` are of different numeric data types.

```python
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 8-45 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 for Python

In this section...

“Python Version Support” on page 8-42
“64-bit or 32-bit Versions of Python and MATLAB” on page 8-43
“Requirements for Building Python from Source” on page 8-43

Python Version Support

To use the MATLAB Engine for Python, you must have a supported version of the reference Python implementation (also known as CPython) installed on your system. The engine supports the following versions:

- version 2.7
- version 3.3
- version 3.4

To download and install Python, see “Install Supported Python Implementation” on page 1-3.

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, you must either add the full path to Python to your PATH environment variable, or 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.
64-bit or 32-bit Versions of Python and MATLAB

The architecture of MATLAB must match the architecture of Python. This means if you run a 64-bit version of Python, you can call only a 64-bit version of MATLAB with the engine. Likewise, if you are running a 32-bit version of Python, you can call only a 32-bit version of MATLAB.

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:

```python
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. (See Cross Platform for more details.)

To test whether your version of MATLAB is 32-bit or 64-bit, from the MATLAB Command Window select Help > About MATLAB or use the `computer` function at the command prompt.

Requirements for Building Python from Source

If you build Python from source on a Linux or Mac system, configure the build with the `--enable-shared` option. The MATLAB Engine for Python must link to the Python shared library. The library builds only if you set the `--enable-shared` option.

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 3.3 or 3.4 on Linux.
Limitations to MATLAB Engine for Python

- 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.
- 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.)
- The MATLAB Engine for Python is not thread-safe.
Troubleshoot MATLAB Errors in Python

In this section...

“MATLAB Errors in Python” on page 8-45
“MatlabExecutionError: Undefined Function” on page 8-45
“SyntaxError: Expression Not Valid Target” on page 8-46
“SyntaxError: Invalid Syntax” on page 8-46

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 traceback and shows the error message only.)

```python
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`.

```python
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 traceback and shows the error message only.)

```python
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
```

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.

```python
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.

```python
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
```

MATLAB and Python functions can have the same name. When this happens, 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 3.3 and 3.4, the above code runs because `print` is a built-in function, not a reserved word.)

To avoid this error, call the MATLAB function with `eval`.

```python
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 3.3 and 3.4 reserved words (https://docs.python.org/3/reference/lexical_analysis.html#keywords)
Using Java Libraries from MATLAB

• “Call Method on Java Object” on page 9-2
• “Java Libraries” on page 9-3
• “Bring Java Classes into MATLAB Workspace” on page 9-5
• “Convert Java String to Uppercase” on page 9-12
• “Use Class in Java JAR File on Static Class Path” on page 9-13
• “Call User-Defined Java Class on Dynamic Class Path” on page 9-14
• “Java Objects” on page 9-15
• “Java Object Methods” on page 9-23
• “Java Arrays” on page 9-31
• “Pass Data to Java Methods” on page 9-48
• “Handle Data Returned from Java Methods” on page 9-58
• “Read URL” on page 9-64
• “Find Internet Protocol Address” on page 9-66
• “Create and Use Phone Book” on page 9-68
• “Java Heap Memory Preferences” on page 9-80
Call Method on Java Object

MATLAB loads Java class definitions from files that are on the Java class path.

Use one of the following syntaxes to call method on Java object obj.

- MATLAB syntax:
  ```matlab
  package.class.method(obj,arg1,...,argn)
  ```
- Java syntax:
  ```java
  obj.method(arg1,...,argn)
  ```
- Use the following when the method name exceeds the maximum length of a MATLAB identifier.
  ```java
  javaMethod(method,obj,arg1,...,argn)
  ```

You do not need to load or import the Java class. Use the MATLAB `import` function to simplify the Java class name in a command.

See Also
import | javaclasspath

More About
- "Java Class Path" on page 9-5
- "Simplifying Java Class Names Using import Function" on page 9-9
Java Libraries

In this section...

“Java Software Is Integral to MATLAB” on page 9-3
“When to Use Java Libraries in MATLAB” on page 9-3
“To Learn More About Java Programming Language” on page 9-4
“Platform Support for JVM Software” on page 9-4

Java Software Is Integral to MATLAB

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

From MATLAB, you can:

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking. For example, the URL class provides convenient access to resources on the Internet.
- Access third-party Java classes
- Easily construct Java objects in MATLAB workspace
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

When to Use Java Libraries in MATLAB

The MATLAB Java interface is intended for MATLAB users who want to take advantage of the special capabilities of the Java programming language.

- You need to access, from MATLAB, the capabilities of available Java classes.
- You are familiar with object-oriented programming in Java or in another language, such as C++.
- You are familiar with the MATLAB Class System, or with “MEX File Creation API”.
To 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.

Platform Support for JVM Software

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

`version -java`

Related Examples

- “Convert Java String to Uppercase” on page 9-12
- “Call Method on Java Object” on page 9-2
Bring Java Classes into MATLAB Workspace

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction” on page 9-5</td>
</tr>
<tr>
<td>“Defining New Java Classes” on page 9-5</td>
</tr>
<tr>
<td>“Java Class Path” on page 9-5</td>
</tr>
<tr>
<td>“Making Java Classes Available in MATLAB Workspace” on page 9-7</td>
</tr>
<tr>
<td>“Loading Java Class Definitions” on page 9-9</td>
</tr>
<tr>
<td>“Simplifying Java Class Names Using import Function” on page 9-9</td>
</tr>
<tr>
<td>“Locating Native Method Libraries” on page 9-10</td>
</tr>
</tbody>
</table>

**Introduction**

You can draw from an extensive collection of existing Java classes or create your own class definitions to use with MATLAB. This topic explains how to find the class definitions that you need or how to create classes of your own design. This topic also describes how to specify the native method libraries used by Java code.

**Defining New Java Classes**

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 Supported and Compatible Compilers website.

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

**Java Class Path**

- “Static Path” on page 9-6
- “Dynamic Path” on page 9-7

MATLAB loads Java class definitions from files that are on the Java class path. The class path is a series of file and folder specifications that MATLAB uses to locate third-party and user-defined class definitions. When loading a Java class, MATLAB searches 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.

MATLAB segments the Java class path into a static path and a dynamic path. MATLAB provides the dynamic path as a convenience for when you develop your own Java classes. After you develop and debug a Java class, add the class to the static path.

To view the two path segments, use the \texttt{javaclasspath} function.

\textbf{Static Path}

Use the static path if you want the class functionality to run the same way it does in Java. Also, the static path offers better class loading performance than the dynamic path.

To add files to the static path, create a \texttt{javaclasspath.txt} file:

1. Create an ASCII text file and name the file \texttt{javaclasspath.txt}.
2. Enter the name of a Java class folder or jar file, one per line. For example:
   \texttt{c:\work\javaclasses}
3. To simplify folder specifications in cross-platform environments, use these macros: \texttt{$matlabroot}, \texttt{$arch}, and \texttt{$jre\_home}.
4. Save the file in your preferences folder. To view the location of the preferences folder, type:
   \texttt{prefdir}

   Alternatively, save the \texttt{javaclasspath.txt} file in your MATLAB startup folder. Classes specified in this file override classes specified in the \texttt{javaclasspath.txt} file in the preferences folder.
5. Restart MATLAB.

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

If you do not want MATLAB to use the entries in the \texttt{javaclasspath.txt} file, start MATLAB with the \texttt{-nouserjavapath} option.

\textbf{Note:} Do not put Java classes on the static path if they have dependencies on classes on the dynamic path.
Bring Java Classes into MATLAB Workspace

**Dynamic Path**

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.

MATLAB always searches the static path before the dynamic path.

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.

The dynamic path offers greater flexibility in changing the path. However, 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 identical to classes on the static path. If your class does not behave as expected, use the static path.

After developing a Java class, put the class on the static path.

**Making Java Classes Available in MATLAB Workspace**

To make third-party and user-defined Java classes available to MATLAB, place them on either the static or dynamic Java class path.

- “Making Individual (Unpackaged) Classes Available” on page 9-7
- “Making Entire Packages Available” on page 9-8
- “Making Classes in a JAR File Available” on page 9-8
- “Loading a Class Using Java `Class.forName` Method” on page 9-8

**Making Individual (Unpackaged) Classes Available**

You can use individual classes (classes that are not part of a package) in MATLAB. To make them available, specify the full path to the folder you want to use for the `.class` files.

For example, to make available your compiled Java classes in the file `d:\work\javaclasses\test.class`, add the following entry to the static or dynamic class path:

```
d:\work\javaclasses
```
Using Java Libraries from MATLAB

To put this folder on the static class path, edit the `javaclasspath.txt` file.

To put this folder on the dynamic class path, use the following command:

```javaaddpath d:\work\javaclasses```

**Making Entire Packages Available**

You can access classes in a package. 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 `d:\work\com\mw\tbx\ini`, add the following folder to your static or dynamic class path:

`d:\work`

**Making Classes in a JAR File Available**

You can use the `jar` (Java Archive) tool to create a JAR file, containing multiple Java classes and packages in a compressed ZIP format. For information on `jar` 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 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 any file name.

For example, to make available the JAR file `e:\java\classes\utilpkg.jar`, add the following file specification to your static or dynamic class path:

```
e:\java\classes\utilpkg.jar
```

You can now call the public methods in the JAR file. For information about these methods, refer to the JAR file documentation.

**Loading a Class Using Java Class.forName Method**

Use the `javaObjectEDT` function instead of the Java `Class.forName` method. For example, replace the following statement:
java.lang.Class.forName('xyz.myapp.MyClass')

with:

javaObjectEDT('xyz.myapp.MyClass')

## Loading Java Class Definitions

Normally, MATLAB loads a Java class automatically when your code first uses it, for example, when you call its constructor. However, be aware of the following exception. When you use the `which` function on methods defined by Java classes, the function only acts on the classes currently *loaded* into the MATLAB workspace. In contrast, `which` always operates on MATLAB classes, whether they are loaded.

### Determining Which Classes Are Loaded

At any time during a MATLAB session, you can obtain a listing of all the Java classes that are currently loaded. To do so, use the `inmem` function as follows:

```matlab
[M,X,J] = inmem
```

This function returns the list of Java classes in the output argument `J`. (It also returns the names of all currently loaded MATLAB functions in `M`, and the names of all currently loaded MEX-files in `X`.)

Here is a sample of output from the `inmem` function:

```matlab
[m,x,j] = inmem;
j
```

MATLAB displays:

```matlab
j =
'java.util.Date'
'com.mathworks.ide.desktop.MLDesktop'
```

### Simplifying Java Class Names Using `import` Function

Your 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 the following statements:

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

Any code that follows these import statements can refer to the String, Frame, and Enumeration classes without using the package names. For example:

```matlab
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:

```matlab
clear import
```

### Locating Native Method Libraries

Java classes can dynamically load native methods using the Java method `java.lang.System.loadLibrary("LibFile")`. In order for the JVM software to locate the specified library file, the folder containing it must be on the Java Library Path. This path is established when the MATLAB runs the JVM software 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:

• Specify each new folder on a line by itself.
• Specify only the folder names, not the names of the DLL files. The `loadLibrary` call reads the file names.

• To simplify the specification of directories in cross-platform environments, use any of these macros: `$matlabroot`, `$arch`, and `$jre_home`.

You also can create a `javalibrarypath.txt` file in your MATLAB startup folder. Libraries specified in this file override libraries specified in the `javalibrarypath.txt` file in the preferences folder.

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

**See Also**

import | inmem | javaclasspath

**Related Examples**

• “Use Class in Java JAR File on Static Class Path” on page 9-13
• “Call User-Defined Java Class on Dynamic Class Path” on page 9-14
Convert Java String to Uppercase

This example shows how to call a built-in Java String class method, `toUpperCase`.

Create a Java string using the built-in class `java.lang.String`.

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

Display methods for the string.

```java
methods(str)
```

Methods for class `java.lang.String`:

- `String` GetBytes replaceAll
- `charAt` getChars replaceFirst
- `codePointAt` getClass split
- `codePointBefore` hashCode startsWith
- `codePointCount` indexOf subSequence
- `compareTo` intern substring
- `compareToIgnoreCase` isEmpty toCharArray
- `concat` lastIndexOf toLowerCase
- `contains` length toString
- `contentEquals` matches toUpperCase
- `copyValueOf` notify trim
- `endsWith` notifyAll valueOf
- `equals` offsetByCodePoints wait
- `equalsIgnoreCase` regionMatches
- `format` replace

Call the `toUpperCase` method.

```java
toUpperCase(str)
```

```java
ans =
HELLO
```
Use Class in Java JAR File on Static Class Path

This example shows how to call a class method in a Java Archive (JAR) file. The example uses a JAR file named `mylibrary.jar` in the folder `C:\Documents\MATLAB\`, containing a method, `package.class.mymethod(params)`. Substitute your own JAR file name and path, and read the documentation to call your own method.

This example puts the JAR file on the static Java class path, making the classes always available in MATLAB. If you only want to use the classes in the current MATLAB session, add the JAR file to the dynamic class path using the `javaaddpath` function.

Copy the `mylibrary.jar` file to the `C:\Documents\MATLAB\` folder.

Add the JAR file to the static class path. Open 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. From now on, the JAR file is available to MATLAB.

Restart MATLAB.

Call the method.

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

See Also

`javaaddpath` | `javaclasspath`

Related Examples

- “Call User-Defined Java Class on Dynamic Class Path” on page 9-14

More About

- “Java Class Path” on page 9-5
Call User-Defined Java Class on Dynamic Class Path

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

This example puts 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. To make the class always available in MATLAB, use the static class path by editing the `javaclasspath.txt` file in your `prefdir` folder.

Add the class to the dynamic Java class path.

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

To access the class, you must modify the Java path every time you start MATLAB.

Call the method.

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

See Also
javaaddpath | javaclasspath

Related Examples

- “Use Class in Java JAR File on Static Class Path” on page 9-13

More About

- “Java Class Path” on page 9-5
Java Objects

**Overview**

You create a Java object in the MATLAB workspace by calling one of the constructors of that class. You then use commands and programming statements to perform operations on these objects. You also can save your Java objects to a MAT-file and, in subsequent sessions, reload them into MATLAB.

**Constructing Java Objects**

You construct Java objects in the MATLAB workspace by calling the Java class constructor, which has the same name as the class. For example, the following constructor creates a `myDate` object:

```plaintext
myDate = java.util.Date

myDate =
Thu Aug 23 12:58:54 EDT 2007
```

MATLAB displays information for your system.

**Using the `javaObjectEDT` Function**

Under certain circumstances, use the `javaObjectEDT` function to construct a Java object. The following syntax invokes the Java constructor for class, `class_name`, with the argument list that matches `x1, ..., xn`, and returns a new object, `J`.

---

In this section...

- “Overview” on page 9-15
- “Constructing Java Objects” on page 9-15
- “Concatenating Java Objects” on page 9-17
- “Saving and Loading Java Objects to MAT-Files” on page 9-18
- “Finding the Public Data Fields of an Object” on page 9-19
- “Accessing Private and Public Data” on page 9-20
- “Determining the Class of an Object” on page 9-21
Using Java Libraries from MATLAB

```matlab
J = javaObjectEDT('class_name',x1,...,xn);
```

For example, to construct and return a Java object of class `java.lang.String`, type:

```matlab
jstr = javaObjectEDT('java.lang.String','hello');
```

With the `javaObjectEDT` function you can:

- Use classes that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify the class for an object at run time, for example, as input from an application user.

The default MATLAB constructor syntax requires that no segment of the input class name can be longer than `namelengthmax` characters. (A class name segment is any portion of the class name before, between, or after a dot. For example, there are three segments in class, `java.lang.String`.) MATLAB truncates any class name segment that exceeds `namelengthmax` characters. In the rare case where you use a class name of this length, use `javaObjectEDT` to instantiate the class.

The `javaObjectEDT` function also allows you to specify the Java class for the object being constructed at run time. In this situation, you call `javaObjectEDT` with a string variable in place of the class name argument.

```matlab
class = 'java.lang.String';
text = 'hello';
jstr = javaObjectEDT(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, type:

```matlab
jstr = java.lang.String('hello');
```

Use the `javaObjectEDT` function instead of the Java `Class.forName` method. For example, replace the following statement:

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

with:

```matlab
javaObjectEDT('xyz.myapp.MyClass')
```
Note: Typically, you do not need to use `javaObjectEDT`. The default MATLAB syntax for instantiating a Java class is simpler and is preferable for most applications. Use `javaObjectEDT` primarily for the previously described cases.

Java Objects Are References in MATLAB Applications

In MATLAB, Java objects are *references* and do not adhere to MATLAB copy-on-assignment and pass-by-value rules. For example:

```matlab
classname = java.util.Date;
setHours(classname,10)
newclass = classname;
```

In this example, the variable `newclass` is a reference to `classname`, not a copy of the object. Any change to the object referenced by `newclass` also changes the object at `classname`. Either MATLAB code or Java code might change the object.

The following example shows that `classname` and `newclass` are references to the same object. When you change the hour via one reference (`newclass`), the change is reflected through the other reference (`classname`), as well.

```matlab
setHours(newclass,8)
getHours(classname)
```

ans =

8

Concatenating Java Objects

You can concatenate Java objects in the same way that you concatenate native MATLAB types. You use either the `cat` function or the `[ ]` operators to tell MATLAB to assemble the enclosed objects into a single object.

Concatenating Objects of the Same Class

If concatenate objects of the same Java class, the concatenation is an array of objects from the same class.

In the following example, the `cat` function concatenates two objects of the class `java.awt.Integer`. The class of the result is also `java.awt.Integer`.

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

ans =
java.lang.Integer[]:
   [88]
   [45]

Concatenating Objects of Unlike Classes

If you concatenate objects of unlike classes, MATLAB finds one class from which all of the input objects inherit. The concatenation is an instance of this class. 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`.

```java
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`, which is the root of the entire Java class hierarchy.

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

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

Saving and Loading Java Objects to MAT-Files

Use the `save` function to save a Java object to a MAT-file. To load it back into MATLAB from that MAT-file, use the `load` function. When you save or load a Java object, the object and its class must meet all of the following criteria.
• The class implements the `Serializable` interface (part of the Java API), either directly or by inheriting it from a parent class. Any embedded or otherwise referenced objects must also implement `Serializable`.

• The definition of the class is not changed between saving and loading the object. Any change to the data fields or methods of a class prevents the loading (deserialization) of an object that was constructed with the old class definition.

• Either the class does not have any transient data fields, or the values in transient data fields of the object to be saved are not significant. Values in transient data fields are never saved with the object.

If you define your own Java classes, or subclasses of existing classes, follow this 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.

**Finding the Public Data Fields of an Object**

To list the public fields that belong to a Java object, use the `fieldnames` function, which takes either of these forms.

```matlab
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Calling `fieldnames` without `-full` returns the names of all the data fields (including inherited) on the object. With the `-full` qualifier, `fieldnames` returns the full description of the data fields defined for the object, including type, attributes, and inheritance information.

For example, create an `Integer` object with the command:

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

To see a full description of the data fields of `value`, type:

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

```matlab
ans =
    'static final int MIN_VALUE'
    'static final int MAX_VALUE'
    'static final java.lang.Class TYPE'
    'static final int SIZE'
```
Accessing Private and Public Data

Java API classes provide accessor methods you can use to read from and, where allowed, to modify private data fields. These methods are sometimes referred to as get and set methods, respectively.

Some Java classes have public data fields, which your code can read or modify directly. To access these fields, use the syntax object.field.

Examples

The java.awt.Frame class provides an example of access to both private and public data fields. This class has the read accessor method getSize, which returns a java.awt.Dimension object. The Dimension object has data fields height and width, which are public and therefore directly accessible. For example, to access this data, type:

```java
frame = java.awt.Frame;
frameDim = getSize(frame);
height = frameDim.height;
frameDim.width = 42;
```

The sample code for “Find Internet Protocol Address” on page 9-66 uses calls to data field accessors on a java.net.InetAddress object.

```java
hostname = address.getHostName;
ipaddress = address.getHostAddress;
```

Accessing Data from a Static Field

In a Java language program, a static data field is a field that applies to an entire class of objects. Static fields are most commonly accessed in relation to the class name itself. For example, the following code accesses the TYPE field of the Integer class using the package and class names, java.lang.Integer.

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

In MATLAB, you can use that same syntax. Or you can refer to the TYPE field as an instance of the class. The following example creates an instance of java.lang.Integer called value, and then accesses the TYPE field using the name value rather than the package and class names.

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

Assigning to a Static Field

Assign values to static fields using the static `set` method of the class. Alternatively, assign values using an instance of the class. For more information, see “Accessing Data from a Static Field” on page 9-20. You can assign `value` to the field `staticFieldName` in the following example by referring to this field as an instance of the class.

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

**Note:** MATLAB does not allow assignment to static fields using the class name itself.

Determining the Class of an Object

To find the class of a Java object, use the query form of the `class` function. After execution of the following example, `myClass` contains the name of the package and class that the object `value` instantiates.

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

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

Because this form of `class` also works on MATLAB objects, it does not, in itself, tell you whether it is a Java class. To determine the type of class, use the `isjava` function, which returns 1 if `obj` is a Java object, and 0 if it is not. For example, type:

```java
isjava(value)
```

```matlab
ans =
1
```

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

```java
isa(value, 'java.lang.Integer')
```
ans =
  1
Java Object Methods

In this section...

“Calling Syntax” on page 9-23
“Obtaining Method Information” on page 9-25
“Java Methods That Affect MATLAB Commands” on page 9-28
“How MATLAB Handles Undefined Methods” on page 9-29
“How MATLAB Handles Java Exceptions” on page 9-30
“Method Execution in MATLAB” on page 9-30

Calling Syntax

To call methods on Java objects, you can use either Java calling syntax or MATLAB calling syntax. Under certain circumstances, use the MATLAB `javaMethod` function.

• “Java Calling Syntax” on page 9-23
• “MATLAB Calling Syntax” on page 9-23
• “Using the javaMethod Function” on page 9-24
• “Calling Syntax for Static Methods of Java Classes” on page 9-24

Java Calling Syntax

To call methods on Java objects, use the Java syntax:

```
object.method(arg1,...,argn)
```

MATLAB Calling Syntax

To call methods on Java objects, use the MATLAB syntax:

```
method(object,arg1,...,argn)
```

For example, to call the `getHours` and `setHours` methods using MATLAB syntax:

```
mlDate = java.util.Date;
setHours(mlDate,3)
getHours(mlDate)
```

ans =
Using the `javaMethod` Function

**Note:** The MATLAB syntax is the preferred syntax for invoking a Java method. Use `javaMethod` for these special cases only.

Use the MATLAB `javaMethod` function to:

- Use Java methods with names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify a Java method to invoke at run time.

For example, your code calls `javaMethod` with a string variable in place of the `method` argument. When you use `javaMethod` to invoke a static method, you also can use a string variable in place of the class name argument.
- Call the constructor of or a static method in an inner class. In the `javaMethod` and `javaObject` functions, specify the class name, using the `$` character, as `OuterClass$InnerClass`.

For example, suppose class `com.ams.MyClass` contains class `MyInnerClass` with static method `methodname`. In Java, the calling syntax is:

```
out = com.ams.MyClass.MyInnerClass.methodname(arg);
```

In MATLAB, type:

```
out = javaMethod('methodname','com.ams.MyClass$MyInnerClass',arg)
```

**Calling Syntax for Static Methods of Java Classes**

To invoke a static method on a Java class, use the Java syntax:

```
class.method(arg1,...,argn)
```

For example, call the static method, `isNaN`:

```
java.lang.Double.isNaN(2.2)
```

```
ans =
    0
```
Obtaining Method Information

MATLAB provides the `methods` and `methodsview` functions to obtain information about the Java methods you are using. You also can request a listing of every Java class that you loaded into MATLAB that implements a specified method.

- “Using methods to Display Method Names and Arguments” on page 9-25
- “Using methodsview to Display Argument Types and Exceptions” on page 9-26
- “Using which to Determine What Classes Define a Method” on page 9-27

Using methods to Display Method Names and Arguments

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.

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.

```matlab
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
Using `methodsview` to Display Argument Types and Exceptions

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.

The following command lists information on all methods in the `java.awt.MenuItem` class. Type:

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

A new window appears, listing one row of information for each method in the class.

![Methods of class java.awt.MenuItem](image-url)
Each row in the window displays up to six fields of information describing the method. The following table lists the fields displayed in the methodsview window along with a description and examples of each field type.

### Fields Displayed in methodsview Window

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

### Using which to Determine What Classes Define a Method

Use the which function to display the fully qualified name (package and class name) of a method implemented by a loaded Java class. To find all classes that define the specified method, use the which function with the -all qualifier.

For example, suppose you want 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 the following:

```
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 `inmem = inmem`, described in “Determining Which Classes Are Loaded” on page 9-9.

For a description of how Java classes are loaded, see “Making Java Classes Available in MATLAB Workspace” on page 9-7.

**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, these objects’ classes. 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 are calling 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 and char**

You can change the output of the MATLAB `double` and `char` functions by defining your own Java methods, `toDouble` and `toChar`. For more information, see “Converting to the MATLAB double Type” on page 9-60 and “Converting to the MATLAB char Type” on page 9-61.

**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. It executes the MATLAB `size` function instead.

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

```matlab
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.
Handling Java Exceptions

Use the matlab.exception.JavaException class to handle Java exceptions.

Method Execution 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.

You, therefore, 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 this is not the case, and the fact that other Java UI code might be already running can lead to problems.
Java Arrays

In this section...

“Introduction” on page 9-31
“How MATLAB Represents the Java Array” on page 9-31
“Creating an Array of Objects in MATLAB” on page 9-35
“Accessing Elements of a Java Array” on page 9-38
“Assigning to a Java Array” on page 9-41
“Concatenating Java Arrays” on page 9-44
“Creating a New Array Reference” on page 9-45
“Creating a Copy of a Java Array” on page 9-46

Introduction

You can pass singular Java objects to and from methods or you can pass them in an array, providing the method expects them in that form. This array must either be a Java array (returned from another method call or created within the MATLAB) or, under certain circumstances, a MATLAB cell array. This section describes how to create and manipulate Java arrays in MATLAB. Later sections describe how to use MATLAB cell arrays in calls to Java methods.

Note: 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, as its individual elements can be indexed using one subscript.

How MATLAB Represents the Java Array

The term Java array refers to any array of Java objects returned from a call to a Java class constructor or method. You can also construct a Java array within MATLAB using the javaArray function. The structure of a Java array is different from the structure of a MATLAB matrix or array. MATLAB hides these differences whenever possible, allowing you to operate on the arrays using the usual MATLAB command syntax. Just the same, keep in mind the following differences as you work with Java arrays.
Representing More Than One Dimension

An array in the Java language is strictly a one-dimensional structure because it is measured only in length. If you want to work with a two-dimensional array, you can create an equivalent structure using an array of arrays. To add further dimensions, you add more levels to the array, making it an array of arrays of arrays, and so on. You can use such multilevel arrays when working in MATLAB, as it is a matrix and array-based programming language.

MATLAB makes it easy for you to work with multilevel Java arrays by treating them like the matrices and multidimensional arrays that are a part of the language itself. You access elements of an array of arrays using the same MATLAB syntax that you use if you are handling a matrix. If you add more levels to the array, MATLAB can access and operate on the structure as if it is a multidimensional MATLAB array.

The left side of the following figure shows Java arrays of one, two, and three dimensions. To the right of each representation is the same array represented in MATLAB. Single-dimension arrays are represented as column vectors.
Java Arrays

Array Indexing

Java array indexing is different than MATLAB array indexing. Java array indices are zero-based, 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 same elements using the MATLAB indexing style of \( y(1) \) through \( y(N) \). Thus, if you have a Java array of 10 elements, the seventh element is obtained using \( y(7) \), and not \( y[6] \) as you use when writing a Java language program.
The Shape of the Java Array

A Java array can be different from a MATLAB array in its overall shape. A two-dimensional MATLAB array maintains a rectangular shape, as each row is of equal length and each column of equal height. The Java counterpart, an array of arrays, 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 terms jagged or ragged are commonly used to describe this arrangement of array elements as the array ends do not match up evenly. When a Java method returns an array with this type of structure, it is stored in a cell array by MATLAB.

Interpreting the Size of a Java Array

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

Determining the size of a Java array of arrays is not so simple. The potentially ragged shape of an array returned from a Java method makes it impossible to size the array in the same way as for a rectangular matrix. In a ragged Java array, there is no single value that represents the size of the lower-level arrays.

When the size function is applied to a Java array of arrays, the resulting value describes the top level of the specified array. For the Java array:

\[
\text{size}(A) = 3 \times 1
\]

\[
\text{size}(A(3)) = 5 \times 1
\]

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, say 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 is useful only for sizing the first dimension, \texttt{dim=1}, the only non-unary dimension.

\[ m = \text{size}(X, \text{dim}) \]
size(A, 1)
ans =
   3

**Interpreting the Number of Dimensions of a Java Arrays**

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

**Creating an Array of Objects in MATLAB**

To call a Java method that has one or more arguments defined as an array of Java objects, you must, under most circumstances, pass your objects in a Java array. You can construct an array of objects in a call to a Java method or constructor. Or you can create the array within MATLAB.

The MATLAB \texttt{javaArray} function lets you create a Java array structure that can be handled in MATLAB 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 an array structure that satisfies the dimensions requested in the \texttt{javaArray} command.

- “Using the javaArray Function” on page 9-36
- “Using MATLAB Syntax” on page 9-36
Using the `javaArray` Function

To create a Java object array, use the MATLAB `javaArray` function, which has the following syntax:

\[ A = \text{javaArray}('\text{element\_class}', m, n, p, \ldots) \]

The first argument is the `\text{element\_class}` string, which names the class of the elements in the array. Specify the fully qualified name (package and class name). The remaining arguments (\(m, n, p, \ldots\)) are the number of elements in each dimension of the array.

An array that you create with `javaArray` is equivalent to the array that you create with the Java code.

\[ A = \text{new element\_class}[m][n][p] \ldots; \]

The following command builds a Java array of four lower-level arrays, each capable of holding five objects of the `java.lang.Double` class.

\[ \text{dblArray} = \text{javaArray}('java.lang.Double',4,5); \]

The `javaArray` function does not deposit any values into the array elements that it creates. You must do this separately. The following MATLAB code stores objects of the `java.lang.Double` type in the Java array `dblArray` that was created.

\[
\begin{align*}
\text{for } & \ m = 1:4 \\
& \quad \text{for } n = 1:5 \\
& \quad \quad \text{dblArray}(m,n) = \text{java.lang.Double}((m*10) + n); \\
& \quad \text{end} \\
& \text{end}
\end{align*}
\]

\[ \text{dblArray} \]

\[
\begin{bmatrix}
\end{bmatrix}
\]

Using MATLAB Syntax

You also can create an array of Java objects using syntax that is more typical to MATLAB. For example, the following syntax creates a 4-by-5 MATLAB array of type `double` and assigns zero to each element of the array.
matlabArr(4,5) = 0;

You use similar syntax to create a Java array in MATLAB, except that you specify the Java class name. The value being assigned, 0 in this example, is stored in the final element of the array, `javaArr(4,5)`. All other elements of the array receive the empty matrix.

`javaArr(4,5) = java.lang.Double(0)`

`javaArr = java.lang.Double[][]:
    [ ]  [ ]  [ ]  [ ]  [ ]
    [ ]  [ ]  [ ]  [ ]  [ ]
    [ ]  [ ]  [ ]  [ ]  [ ]
    [ ]  [ ]  [ ]  [ ]  [0]

**Note:** You cannot change the number of dimensions of an existing Java array as you can with a MATLAB array. The same restriction exists when working with Java arrays in the Java language. See the following example.

This example first creates a scalar MATLAB array, and then changes it to a 2D array.

`matlabArr = 0;`  
`matlabArr(4,5) = 0`  

`matlabArr = `  
``
  0  0  0  0  0  
  0  0  0  0  0  
  0  0  0  0  0  
  0  0  0  0  0  
``

When you try this technique with a Java array, you get an error message.

`javaArr = java.lang.Double(0);`  
`javaArr(4,5) = java.lang.Double(0);`

*Index exceeds Java array dimensions.*

Similarly, you cannot create an array of Java arrays from a Java array.
Accessing Elements of a Java Array

- “MATLAB Array Indexing” on page 9-38
- “Single Subscript Indexing” on page 9-38
- “Colon Operator Indexing” on page 9-39
- “Using END in a Subscript” on page 9-40
- “Elements of java.lang.Object Arrays Are Converted to MATLAB Types” on page 9-41

MATLAB Array Indexing

You can access elements of a Java object array by using the MATLAB array indexing syntax, \texttt{A(row,col)}. For example, to access the element of array \texttt{dblArray} at row 3, column 4, use:

\begin{verbatim}
row3_col4 = dblArray(3,4)
\end{verbatim}

\begin{verbatim}
row3_col4 =
34.0
\end{verbatim}

To access this element in a Java language program, use \texttt{dblArray[2][3]}.

You also can use MATLAB array indexing syntax to access an element in the data field of an object. Suppose that \texttt{myMenuObj} is an instance of a window menu class. This user-supplied class has a data field, \texttt{menuItemArray}, which is a Java array of \texttt{java.awt.menuItem}. To get element 3 of this array, use the following command.

\begin{verbatim}
currentItem = myMenuObj.menuItemArray(3)
\end{verbatim}

Single Subscript Indexing

Elements of a MATLAB matrix are most commonly referenced using both row and column subscripts. For example, you use \texttt{x(3,4)} to reference the array element at the intersection of row 3 and column 4. Sometimes it is more advantageous to use just a single subscript. MATLAB provides this capability (see the section on “Linear Indexing” in MATLAB Mathematics).

Indexing into a MATLAB matrix using a single subscript references one element of the matrix. Using the MATLAB matrix shown here, \texttt{matlabArr (3)} returns a single element of the matrix.

\begin{verbatim}
matlabArr = [11 12 13 14 15; 21 22 23 24 25; ...]
\end{verbatim}
Java Arrays

31 32 33 34 35; 41 42 43 44 45]

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

matlabArr(3)
ans =
  31

Indexing this way into a Java array of arrays references an entire subarray of the overall structure. Using the dblArray Java array, that looks the same as matlabArr, dblArray(3) returns the 5-by-1 array that makes up the entire third row.

row3 = dblArray(3)
row3 =
  java.lang.Double[]:
    [31]
    [32]
    [33]
    [34]
    [35]

This feature allows you to specify an entire array from a larger array structure, and then manipulate it as an object.

Colon Operator Indexing

Use of the MATLAB colon operator (:) is supported in subscripting Java array references. This operator works just the same as when referencing the contents of a MATLAB array. Using the Java array of java.lang.Double objects shown here, the statement dblArray(2,2:4) refers to a portion of the lower-level array, dblArray(2). A new array, row2Array, is created from the elements in columns 2 through 4.

dblArray
dblArray =
  java.lang.Double[][]:
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
row2Array = dblArray(2,2:4)

row2Array =
java.lang.Double[]:
  [22]
  [23]
  [24]

You also can use the colon operator in single-subscript indexing, as covered in “Single Subscript Indexing” on page 9-38. By making your subscript a colon rather than a number, you can convert an array of arrays into one linear array. The following example converts 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]
  .
  .
  .

This method also works on an N-dimensional Java array structure. Using the colon operator as a single subscripted index into the array produces a linear array composed of all of the elements of the original array.

**Note:** Java and MATLAB arrays are stored differently in memory. This is reflected in the order they are given in a linear array. Java array elements are stored in an order that matches the rows of the matrix (linearArray elements 11, 12, 13, ...). MATLAB array elements are stored in an order that matches the columns (elements 11, 21, 31, ...).

**Using END in a Subscript**

You can use the `end` keyword in the first subscript of an access statement. The first subscript references the top-level array in a multilevel Java array structure.
Note: Using end on lower-level arrays is not valid due to the potentially ragged nature of these arrays (see “The Shape of the Java Array” on page 9-34). In this case, there is no consistent end value to be derived.

The following example displays data from the third to the last row of Java array dblArray.

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

```
last2rows =
java.lang.Double[][]:
    [31] [32] [33] [34] [35]
    [41] [42] [43] [44] [45]
```

Elements of `java.lang.Object` Arrays Are Converted to MATLAB Types

When you access an element of a `java.lang.Object` array, MATLAB converts the element to a MATLAB type, according to the table in “Conversion of Java Object Return Types” on page 9-59. 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, MATLAB converts the element to MATLAB `double`. But MATLAB does not convert a `java.lang.Double` element in a `java.lang.Double` array; MATLAB returns it as `java.lang.Double`.

Assigning to a Java Array

- “Using Single Subscript Indexing for Array Assignment” on page 9-42
- “Assigning to a Linear Array” on page 9-42
- “Assigning the Empty Matrix” on page 9-43
- “Subscripted Deletion” on page 9-43

You assign values to objects in a Java array in essentially the same way as you do in a MATLAB array. Although Java and MATLAB arrays are structured differently, you use the same command syntax to specify which elements you want to assign to. See “Introduction” on page 9-31 for more information on Java and MATLAB array differences.

The following example assigns the value 300 in the `dblArray` element at row 3, column 2. In a Java language program, this is `dblArray[2][1]`. 
Using Java Libraries from MATLAB

dblArray(3,2) = java.lang.Double(300)

myMenuObj.menuItemArray(3) = java.lang.String('Save As...');

Using Single Subscript Indexing for Array Assignment

You can use a single-array subscript to index into a Java array structure that has more than one dimension. Refer to “Single Subscript Indexing” on page 9-38 for a description of this feature as used with Java arrays.

You can use single-subscript indexing to assign values to an array as well. The following example assigns a one-dimensional Java array, onedimArray, to a row of a two-dimensional Java array, dblArray. Start out by creating the one-dimensional array.

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

Since dblArray(3) refers to the 5-by-1 array displayed in the third row of dblArray, you can assign the entire, similarly dimensioned, 5-by-1 onedimArray to it.

dblArray(3) = onedimArray

Assigning to a Linear Array

You can assign a value to every element of a multidimensional Java array by treating the array structure as if it were a single linear array. This entails replacing the single,
numerical subscript with the MATLAB colon operator. If you start with the `dblArray` array, you can initialize the contents of every object in the two-dimensional array with the following statement.

```java
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]
```

You can use the MATLAB colon operator as you would when working with MATLAB arrays. The following statements assign given values to each of the four rows in the Java array, `dblArray`. Remember that each row actually represents a separate Java array in itself.

```java
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]
```

**Assigning the Empty Matrix**

When working with MATLAB arrays, you can assign the empty matrix, (that is, the 0-by-0 array denoted by `[]`) to an element of the array. For Java arrays, you also can assign `[]` to array elements. This stores the null value, rather than a 0-by-0 array, in the Java array element.

**Subscripted Deletion**

When you assign the empty matrix value to an entire row or column of a MATLAB array, you find that MATLAB actually removes the affected row or column from the array. In the example below, the empty matrix is assigned to all elements of the fourth column in the MATLAB matrix, `matlabArr`. Thus, the fourth column is eliminated from the matrix. This changes its dimensions from 4-by-5 to 4-by-4.
matlabArr = [11 12 13 14 15; 21 22 23 24 25; ... 31 32 33 34 35; 41 42 43 44 45]

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

matlabArr(:,4) = []

matlabArr =
11 12 13 15
21 22 23 25
31 32 33 35
41 42 43 45

You can assign the empty matrix to a Java array, but the effect is different. The next example shows that, when the same operation is performed on a Java array, the structure is not collapsed; it maintains its 4-by-5 dimensions.

dblArray(:,4) = []

dblArray =
java.lang.Double[][]:
[125] [125] [125] [ ] [125]
[250] [250] [250] [ ] [250]
[375] [375] [375] [ ] [375]
[500] [500] [500] [ ] [500]

The dblArray data structure is actually an array of five-element arrays of java.lang.Double objects. The empty array assignment placed the null value in the fourth element of each of the lower-level arrays.

**Concatenating Java Arrays**

You can concatenate arrays of Java objects in the same way as arrays of other types. Java objects, however, can only be catenated along the first or second axis. To understand how scalar Java objects are concatenated in MATLAB, see “Concatenating Java Objects” on page 9-17.

Use either the `cat` function or the square bracket ([ ]) operators. This example horizontally concatenates two Java arrays: `d1` and `d2`. 
% Construct a 2-by-3 array of java.lang.Double.
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[][]:
[3] [5] [7]

% Construct a 2-by-2 array of java.lang.Double.
d2 = javaArray('java.lang.Double',2,2);
for m = 1:2
    for n = 1:2
        d2(m,n) = java.lang.Double((n+3)*2 + m-1);
    end
end
d2

d2 =
java.lang.Double[][]:
[8] [10]
[9] [11]

% Concatenate the two along the second dimension.
d3 = cat(2,d1,d2)
d3 =
java.lang.Double[][]:

Creating a New Array Reference

Because Java arrays in MATLAB are references, assigning an array variable to another variable results in a second reference to the array.

Consider the following example where two separate array variables reference a common array. The original array, origArray, is created and initialized. The statement
newArrayRef = origArray creates a copy of this array variable. Changes made to the array referred to by newArrayRef also show up in the original array.

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]

% Make a copy of the array reference
newArrayRef = origArray;
newArrayRef(3,:) = java.lang.Double(0);

origArray

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

Creating a Copy of a Java Array

You can create an entirely new array from an existing Java array by indexing into the array to describe a block of elements, or subarray, and assigning this subarray to a variable. The assignment copies the values in the original array to the corresponding cells of the new array.

As with the example in section “Creating a New Array Reference” on page 9-45, an original array is created and initialized. But, this time, a copy is made of the array contents rather than copying the array reference. Changes made using the reference to the new array do not affect the original.

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]

% ----- Make a copy of the array contents -----
newArray = origArray(:,:);
newArray(3,:) = java.lang.Double(0);
origArray
origArray = java.lang.Double[][]:
    [11] [12] [13] [14]
    [21] [22] [23] [24]
    [31] [32] [33] [34]
Pass Data to Java Methods

**In this section...**

<table>
<thead>
<tr>
<th>“Introduction” on page 9-48</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Conversion of MATLAB Argument Data” on page 9-48</td>
</tr>
<tr>
<td>“Passing Built-In Types” on page 9-50</td>
</tr>
<tr>
<td>“Converting Numbers to Integer Arguments” on page 9-51</td>
</tr>
<tr>
<td>“Passing String Arguments” on page 9-52</td>
</tr>
<tr>
<td>“Passing Java Objects” on page 9-52</td>
</tr>
<tr>
<td>“Other Data Conversion Topics” on page 9-55</td>
</tr>
<tr>
<td>“Passing Data to Overloaded Methods” on page 9-56</td>
</tr>
</tbody>
</table>

**Introduction**

When you call a Java method, MATLAB types you pass in the call are converted to types native to the Java language. MATLAB performs this conversion on each argument that is passed, except for those arguments that are already Java objects. This section describes the conversion that is performed on specific MATLAB types and, at the end, also looks at how argument types affect calls made to overloaded methods.

If the method returns data, MATLAB converts the data to the appropriate MATLAB format. This process is covered in “Handle Data Returned from Java Methods” on page 9-58.

**Conversion of MATLAB Argument Data**

MATLAB converts MATLAB data, passed as arguments to Java methods, into types that best represent the data to the Java language. The following table shows all of the MATLAB base types for passed arguments and the Java types defined for the parameters. Each row 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 all be scalar (1-by-1) arrays or matrices. All of the Java types can be scalar values or arrays.

**Conversion of MATLAB Types to Java Types**
Pass Data to Java Methods

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Java Parameter Type (Scalar or Array)</th>
<th>Closest Type &lt;———&gt; Least Close Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>boolean byte short int long float double</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double float long int short byte boolean</td>
<td></td>
</tr>
<tr>
<td>single</td>
<td>float double</td>
<td></td>
</tr>
<tr>
<td>char (1-by-1 scalar)</td>
<td>String char</td>
<td></td>
</tr>
<tr>
<td>char (1-by-n or n-by-1, n&gt;1)</td>
<td>String char[]</td>
<td></td>
</tr>
<tr>
<td>char(m-by-n, m,n&gt;1)</td>
<td>String[]</td>
<td></td>
</tr>
<tr>
<td>uint8 int8</td>
<td>byte short int long float double</td>
<td></td>
</tr>
<tr>
<td>uint16 int16</td>
<td>short int long float double</td>
<td></td>
</tr>
<tr>
<td>uint32 int32</td>
<td>int long float double</td>
<td></td>
</tr>
<tr>
<td>uint64 int64</td>
<td>long float double</td>
<td></td>
</tr>
<tr>
<td>cell array of strings</td>
<td>String[]</td>
<td></td>
</tr>
<tr>
<td>Java object</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>cell array of object</td>
<td>Object[]</td>
<td></td>
</tr>
<tr>
<td>MATLAB object</td>
<td>Unsupported</td>
<td></td>
</tr>
</tbody>
</table>

For more information about type conversion of arguments passed to Java code, see:

- “Passing Built-In Types” on page 9-50
- “Passing String Arguments” on page 9-52
- “Passing Java Objects” on page 9-52
Passing Built-In Types

The Java language has eight types that are intrinsic to the language and are not represented as Java objects. These are often referred to as built in, or elemental, types and they include boolean, byte, short, long, int, double, float, and char. MATLAB converts its own types to these Java built-in types according to the table, Conversion of MATLAB Types to Java Types. Built-in types are in the first 10 rows of the table.

When a Java method you are calling expects one of these types, you can pass it the type of MATLAB argument shown in the leftmost column of the table. If the method takes an array of one of these types, you can pass a MATLAB array of the type. MATLAB converts the type of the argument to the type assigned in the method declaration.

The following MATLAB code creates a top-level window frame and sets its dimensions. The call to setBounds passes four MATLAB scalars of the double type to the inherited Java Frame method, setBounds, that takes four arguments of the int type. MATLAB converts each 64-bit double type to a 32-bit integer before calling the method.

The setBounds method declaration:

public void setBounds(int x, int y, int width, int height)

The MATLAB code that calls the method:

frame = java.awt.Frame;
setBounds(frame,200,200,800,400)
setVisible(frame,1)

Passing Built-In Types in an Array

To call a Java method with an argument defined as an array of a built-in type, you can create and pass a MATLAB matrix with a compatible base type. The following code defines a polygon by sending four x and y coordinates to the Polygon constructor. Two 1-by-4 MATLAB arrays of double are passed to java.awt.Polygon, which expects integer arrays in the first two arguments. Shown here is the Java method declaration followed by MATLAB code that calls the method, and then verifies the set coordinates.

public Polygon(int xpoints[], int ypoints[], int npoints)

poly = java.awt.Polygon([14 42 98 124], [55 12 -2 62], 4);
[poly.xpoints poly.ypoints] % Verify the coordinates
MATLAB Arrays Are Passed by Value

Since MATLAB arrays are passed by value, any changes that a Java method makes to them are not visible to your MATLAB code. To access changes that a Java method makes to an array, then, rather than passing a MATLAB array, create and pass a Java array, which is a reference. For a description of using Java arrays in MATLAB, see “Java Arrays” on page 9-31.

Note: Generally, it is preferable to have methods return data that has been modified using the return argument mechanism as opposed to passing a reference to that data in an argument list.

Converting 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 (does not fit into the number of bits of the parameter type), MATLAB discards all lowest \( n \) bits, where \( n \) is the number of bits in the parameter type. This is unlike 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 floating point, MATLAB does not convert to an integer in the same manner as Java. A floating point number is first converted to a 64-bit signed integer with the fractional part truncated and then processed as if it were an int64 argument.

When a floating point number is too large to be represented in a 64-bit integer (outside the range from \(-2^{63} - 2^{63}\)), MATLAB converts the value as follows:

- Convert int, short, and byte parameter values to 0.
- Convert long parameter values to java.lang.Long.MIN_VALUE.
- Convert Inf and -Inf values to -1.
- Convert NaN values to 0.
Passing String Arguments

To call a Java method that has an argument defined as an object of class `java.lang.String`, you can pass either a `String` object that was returned from an earlier Java call or a MATLAB 1-by-n character array. If you pass the character array, MATLAB converts the array to a Java object of `java.lang.String` for you.

For a programming example, see “Read URL” on page 9-64. This example shows a MATLAB character array that holds a URL being passed to the Java `URL` class constructor. The following constructor expects a Java `String` argument.

```java
public URL(String spec) throws MalformedURLException
```

In the MATLAB call to this constructor, a character array specifying the URL is passed. MATLAB converts this array to a Java `String` object before calling the constructor.

```matlab
url = java.net.URL('http://archive.ncsa.uiuc.edu/demoweb/');
```

Passing Strings in an Array

When the method you are calling expects an argument of an array of type `String`, you can create such an array by packaging the strings together in a MATLAB cell array. The strings can be of varying lengths since you are storing them in different cells of the array. As part of the method call, MATLAB converts the cell array to a Java array of `String` objects.

In the following example, the `echoPrompts` method of a user-written class accepts a string array argument that MATLAB converted from its original format as a cell array of strings. The parameter list in the Java method appears as follows:

```java
public String[] echoPrompts(String s[])
```

You create the input argument by storing both strings in a MATLAB cell array. MATLAB converts this structure to a Java array of `String`.

```matlab
myaccount.echoPrompts({'Username: ','Password: '})
ans =
    'Username: '
    'Password: '
```

Passing Java Objects

When calling a method that has a parameter 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 automatically convert MATLAB types to Java Object types. Unlike Java, MATLAB does not do autoboxing for numbers, such as converting double to Double.

In the example below, the add method belonging to the java.awt.Menu class requires, as an argument, an object of the java.awt.MenuItem class. The method declaration is:

```java
public MenuItem add(MenuItem mi)
```

The example operates on the frame created in the previous example in “Passing Built-In Types” on page 9-50. The second, third, and fourth lines of code shown here add items to a menu to be attached to the existing window frame. In each of these calls to menu1.add, an object that is an instance of the java.awt.MenuItem Java class is passed.

```java
menu1 = java.awt.Menu('File Options');
menu1.add(java.awt.MenuItem('New'));
menu1.add(java.awt.MenuItem('Open'));
menu1.add(java.awt.MenuItem('Save'));
```

```java
menuBar=java.awt.MenuBar;
menuBar.add(menu1);
frame.setMenuBar(menuBar);
```

**Handling Objects of Class java.lang.Object**

A special case exists when the method being called 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. When passing a MATLAB argument, the argument is automatically converted to the closest Java Object type, which might include Java-style autoboxing, as shown in the following table.

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Java Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>Boolean</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>single</td>
<td>Float</td>
</tr>
<tr>
<td>char (1-by-1 scalar)</td>
<td>Character</td>
</tr>
<tr>
<td>char (1-by-n or n-by-1, n&gt;1)</td>
<td>String</td>
</tr>
<tr>
<td>uint8</td>
<td>Byte</td>
</tr>
<tr>
<td>int8</td>
<td></td>
</tr>
</tbody>
</table>
MATLAB Argument    | Java Object
---------------------|-------------------
uint16               | Short
int16               | Short
uint32               | Integer
int32               | Integer
uint64               | Long
int64               | Long
cell array of strings | String[]
Java object         | Object
cell array of object | Object[]
MATLAB object       | Unsupported

**Passing Objects in an Array**

The only types of object arrays that you can pass to Java methods are Java arrays and MATLAB cell arrays. MATLAB automatically converts the cell array elements to `java.lang.Object` class objects. In order for a cell array to be passed from MATLAB, the corresponding argument in the Java method signature must specify `java.lang.Object` or an array of `java.lang.Object`.

If the objects are already in a Java array, either an array returned from a Java constructor or constructed in MATLAB by the `javaArray` function, then you simply pass it as the argument to the method being called. MATLAB does not convert the argument, because the argument is already a Java array.

The following example shows the `mapPoints` method of a user-written class accepting an array of `java.awt.Point` objects. The declaration for this method is:

```java
public Object mapPoints(java.awt.Point p[])
```

The following MATLAB code creates a 4-by-1 array containing four Java `Point` objects. When the array is passed to the `mapPoints` method, no conversion is necessary because the `javaArray` function created a Java array of `java.awt.Point` objects.

```matlab
pointObj = javaArray('java.awt.Point',4);
pointObj(1) = java.awt.Point(25,143);
pointObj(2) = java.awt.Point(31,147);
pointObj(3) = java.awt.Point(49,151);
pointObj(4) = java.awt.Point(52,176);
```
testData.mapPoints(pointObj);

**Handling a Cell Array of Java Objects**

You create a cell array of Java objects by using the MATLAB syntax \{a_1,a_2,\ldots\}. You index into a cell array of Java objects in the usual way, with the syntax \texttt{a\{m,n,\ldots\}}.

The following example creates a cell array of two \texttt{Frame} objects, \texttt{frame1} and \texttt{frame2}, and assigns it to variable \texttt{frameArray}.

```java
frame1 = java.awt.Frame('Frame A');
frame2 = java.awt.Frame('Frame B');

frameArray = \{frame1, frame2\}
```

```java
frameArray =
[1x1 java.awt.Frame]    [1x1 java.awt.Frame]
```

The next statement assigns element \{1,2\} of the cell array \texttt{frameArray} to variable \texttt{f}.

```java
f = frameArray \{1,2\}
```

```java
f =
java.awt.Frame[frame2,0,0,0x0,invalid,hidden,layout =
java.awt.BorderLayout,resizable,title=Frame B]
```

**Other Data Conversion Topics**

There are several remaining items of interest regarding the way MATLAB converts its data to a compatible Java type. This includes how MATLAB matches array dimensions, and how it handles empty matrices and empty strings.

**How Array Dimensions Affect Conversion**

The term \textit{dimension} refers to the number of subscripts required to address the elements of an array. For example, a 5-by-1 array has one dimension, because you can index individual elements using only one array subscript.

In converting MATLAB to Java arrays, MATLAB handles dimension in a special manner. For a MATLAB array, dimension can be considered as the number of nonsingleton dimensions in the array. For example, a 10-by-1 array has dimension 1, and a 1-by-1 array has dimension 0. In Java code, dimension is determined solely by the
number of nested arrays. For example, \texttt{double[][]} has dimension 2, and \texttt{double} has dimension 0.

If the number of dimensions of the Java array exactly matches the MATLAB array's number of dimensions \( n \), the conversion results in a Java array with \( n \) dimensions. If the Java array has fewer than \( n \) dimensions, the conversion drops singleton dimensions, starting with the first one, until the number of remaining dimensions matches the number of dimensions in the Java array.

**Empty Matrices and Nulls**

An empty matrix is converted to a Java \texttt{null}. The empty string ("\"") in MATLAB translates into an empty (not \texttt{null}) \texttt{String} object in Java code. Empty (0-length) Java arrays remain unchanged.

**Passing Data to Overloaded Methods**

When you invoke an overloaded method on a Java object, MATLAB determines which method to invoke by comparing the arguments your call passes to the arguments defined for the methods. In this discussion, the term \textit{method} includes constructors. When it determines the method to call, MATLAB converts the calling arguments to Java method types according to Java conversion rules, except for conversions involving objects or cell arrays. See “Passing Objects in an Array” on page 9-54.

**How MATLAB Determines the Method to Call**

When your MATLAB function calls a Java 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 Java type defined for the method.

If all of 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. First, MATLAB rejects all methods that have any argument types that are incompatible with the passed arguments.
(for example, if the method has a double argument and the passed argument is a char type).

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 the fitness of the base type minus the difference between the MATLAB array dimension and the Java array dimension. (Array dimensionality is explained in “How Array Dimensions Affect Conversion” on page 9-55.) If two methods have the same fitness, the first one defined in the Java class is chosen.

Example — Calling an Overloaded Method

Suppose that a function constructs a java.io.OutputStreamWriter object, osw, and then invokes a method on the object.

```java
osw.write('Test data', 0, 9);
```

MATLAB finds that the class java.io.OutputStreamWriter defines three write methods.

```java
public void write(int c);
public void write(char[] cbuf, int off, int len);
public void write(String str, int off, int len);
```

MATLAB rejects the first write method, because it takes only one argument. Then, MATLAB assesses the fitness of the remaining two write methods. These differ only in their first argument, as explained below.

In the first of these two write methods, the first argument is defined with base type, char. The table, Conversion of MATLAB Types to Java Types, shows that for the type of the calling argument (MATLAB char), Java type, char, has a value of 6. There is no difference between the dimension of the calling argument and the Java argument. So the fitness value for the first argument is 6.

In the other write method, the first argument has Java type String, which has a fitness value of 7. The dimension of the Java argument is 0, so the difference between it and the calling argument dimension is 1. Therefore, the fitness value for the first argument is 6.

Because the fitness value of those two write methods is equal, MATLAB calls the one listed first in the class definition, with char[] first argument.
Handle Data Returned from Java Methods

**In this section...**

“Introduction” on page 9-58
“Conversion of Java Return Types” on page 9-58
“Conversion of Java Object Return Types” on page 9-59
“Built-In Types” on page 9-59
“Converting Objects to MATLAB Types” on page 9-60

**Introduction**

In many cases, data returned from a Java method is incompatible with the types operated on in the MATLAB platform. When this is the case, MATLAB converts the returned value to a type native to the MATLAB language. This section describes the conversion performed on the various types that can be returned from a call to a Java method.

**Conversion of Java Return Types**

The following table lists Java return types and the resulting MATLAB types. For some Java base return types, MATLAB treats scalar and array returns differently. For information about java.lang.Object type, see “Conversion of Java Object Return Types” on page 9-59.

**Conversion of Java Types to MATLAB Types**

<table>
<thead>
<tr>
<th>Java Return Type</th>
<th>If Scalar Return, Resulting MATLAB Type</th>
<th>If Array Return, Resulting MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>logical</td>
<td>logical</td>
</tr>
<tr>
<td>byte</td>
<td>double</td>
<td>int8</td>
</tr>
<tr>
<td>short</td>
<td>double</td>
<td>int16</td>
</tr>
<tr>
<td>int</td>
<td>double</td>
<td>int32</td>
</tr>
<tr>
<td>long</td>
<td>double</td>
<td>int64</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td>single</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>double</td>
</tr>
</tbody>
</table>
Handle Data Returned from Java Methods

<table>
<thead>
<tr>
<th>Java Return Type</th>
<th>If Scalar Return, Resulting MATLAB Type</th>
<th>If Array Return, Resulting MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char</td>
<td>char</td>
</tr>
</tbody>
</table>

**Note:** MATLAB converts rectangular Java arrays to arrays of the resulting type. When Java returns a nonrectangular (jagged) array, MATLAB converts it to a cell array. For more information, see “How MATLAB Represents the Java Array” on page 9-31.

**Conversion of Java Object Return Types**

When a method call returns data of type `java.lang.Object`, MATLAB converts its value, depending on its actual type, according to the following table. Other object types, including `Object` arrays, are not converted; they remain Java objects. However, if you index into a returned `Object` array, the value will be converted according to the table. For more information, see “Elements of `java.lang.Object` Arrays Are Converted to MATLAB Types” on page 9-41.

**Conversion of `java.lang.Object` Type to MATLAB Types**

<table>
<thead>
<tr>
<th>Actual Java Type</th>
<th>Return Scalar MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Boolean</code></td>
<td>logical</td>
</tr>
<tr>
<td><code>java.lang.Byte</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Short</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Integer</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Long</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Float</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Double</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Character</code></td>
<td>char</td>
</tr>
<tr>
<td><code>java.lang,String</code></td>
<td>char</td>
</tr>
</tbody>
</table>

**Built-In Types**

Java **built-in** types are described in “Passing Built-In Types” on page 9-50. This type includes `boolean, byte, short, long, int, double, float, and char`. When the value
returned from a method call is one of these types, MATLAB converts it according to the table Conversion of Java Types to MATLAB Types.

A single numeric or boolean value converts to a 1-by-1 matrix of double, which is convenient for use in MATLAB. An array of a numeric or boolean return values converts each member of the array to the closest base type to minimize the required storage space. Array member conversions are listed in the right-hand column of the table.

A return value of Java type char converts to a 1-by-1 matrix of char. An array of Java char converts to a MATLAB array of that type.

**Converting Objects to MATLAB Types**

Except for objects of class Object, MATLAB does not convert Java objects returned from method calls to a native MATLAB type. If you want to convert Java object data to a form more readily usable in MATLAB, there are a few MATLAB functions that enable you to do this. These are described in the following sections.

- “Converting to the MATLAB double Type” on page 9-60
- “Converting to the MATLAB char Type” on page 9-61
- “Converting to a MATLAB Structure” on page 9-61
- “Converting to a MATLAB Cell Array” on page 9-62

**Converting to the MATLAB double Type**

Using the double function in MATLAB, you can convert any Java object or array of objects to the MATLAB double type. The action taken by the double function depends on the class of the object you specify:

- If the object is an instance of a numeric class (java.lang.Number or one of the classes that inherit from that class), MATLAB uses a preset conversion algorithm to convert the object to a MATLAB double.
- If the object is not an instance of a numeric class, MATLAB checks the class definition to see if it implements a method called toDouble. MATLAB uses toDouble to perform its conversion of Java objects to the MATLAB double type. If such a method is implemented for this class, MATLAB executes it to perform the conversion.
- If you are using a class of your own design, you can write your own toDouble method to perform conversions on objects of that class to a MATLAB double. This enables you to specify your own means of type conversion for objects belonging to your own classes.
Handle Data Returned from Java Methods

**Note:** If the class of the specified object is not `java.lang.Number`, does not inherit from that `java.lang.Number`, and does not implement a `toDouble` method, then an attempt to convert the object using the `double` function results in a MATLAB error message.

The syntax for the `double` command is as follows, where `object` is a Java object or Java array of objects:

```matlab
double(object);
```

**Converting to the MATLAB char Type**

With the MATLAB `char` function, you can convert `java.lang.String` objects and arrays to MATLAB character arrays.

The syntax for the `char` command is as follows, where `object` is a Java object or Java array of objects:

```matlab
char(object);
```

If the object specified in the `char` command is not an instance of the `java.lang.String` class, MATLAB checks its class to see if it implements a method named `toChar`. If so, MATLAB executes the `toChar` method of the class to perform the conversion. If you write your own class definitions, you can use this feature by writing a `toChar` method that performs the conversion according to your own needs.

**Note:** If the class of the specified object is not `java.lang.String` and it does not implement a `toChar` method, an attempt to convert the object using the `char` function results in a MATLAB error message.

**Converting to a MATLAB Structure**

Java objects are similar to the MATLAB `struct` type in that many of an object's characteristics are accessible via field names defined within the object. You might want to convert a Java object into a MATLAB `struct` to facilitate the handling of its data in MATLAB. Use the MATLAB `struct` function to do this.

The syntax for the `struct` command is as follows, where `object` is a Java object or a Java array of objects:
struct(object);

The following example converts a java.awt.Polygon object into a MATLAB struct. You can access the fields of the object directly using MATLAB struct operations. The last line indexes into the array, pstruct.xpoints, to deposit a new value into the third array element.

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

pstruct = 
npoints: 4
  xpoints: [4x1 int32]
  ypoints: [4x1 int32]

pstruct.xpoints

ans =
  14
  42
  98
  124

pstruct.xpoints(3) = 101;

Converting to a MATLAB Cell Array

Use the cell function to convert a Java array or Java object into a MATLAB cell array. Elements of the resulting cell array are of the MATLAB type (if any) closest to the Java array elements or Java object.

The syntax for the cell command is as follows, where object is a Java object or a Java array of objects.

cell(object);

The following example uses the cell command to create a MATLAB cell array in which each cell holds an array of a different type.

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 to cell arrays
cellArray = {cell(dblArray), cell(ptArray), cell(strArray)}
cellArray =
    {1x10 cell}    {3x1 cell}    {2x2 cell}
cellArray{1,1}  % Array of type double
ans =
    [7]   [14]   [21]   [28]   [35]   [42]   [49]   [56]   [63]   [70]
cellArray{1,2}  % Array of type Java.awt.Point
ans =
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]
cellArray{1,3}  % Array of type char array
ans =
    'one'    'two'
    'three'  'four'
Read URL

In this section...

“Overview” on page 9-64
“Description of URLdemo” on page 9-64
“Running the Example” on page 9-65

Overview

This program, URLdemo, opens a connection to a website specified by a URL (Uniform Resource Locator) to read text from a file at that site.

URLdemo constructs an object of the Java API class, java.net.URL, which enables convenient handling of URLs. Then, it calls a method on the URL object, to open a connection.

To read and display the lines of text at the site, URLdemo uses classes from the Java I/O package java.io. It creates an InputStreamReader object, and then uses that object to construct a BufferedReader object. Finally, it calls a method on the BufferedReader object to read the specified number of lines from the site.

Description of URLdemo

The major tasks performed by URLdemo are:

1 Construct a URL object.

   The example first calls a constructor on java.net.URL and assigns the resulting object to variable url. The URL constructor takes a single argument, the name of the URL to be accessed, as a string. The constructor checks whether the input URL has a valid form.

   url = java.net.URL('http://www.mathworks.com')

2 Open a connection to the URL.

   The second statement of the example calls the method, openStream, on the URL object url, to establish a connection with the website named by the object. The method returns an InputStream object to variable, is, for reading bytes from the site.
is = openStream(url);

3  Set up a buffered stream reader.

The next two lines create a buffered stream reader for characters. The
java.io.InputStreamReader constructor is called with the input stream
is, to return to variable isr an object that can read characters. Then, the
java.io.BufferedReader constructor is called with isr, to return a
BufferedReader object to variable br. A buffered reader provides for efficient
reading of characters, arrays, and lines.

isr = java.io.InputStreamReader(is);
br = java.io.BufferedReader(isr);

4  Read and display lines of text.

The following statements read the lines of HTML text from the site, looking for
text beginning with a paragraph tag, <p>. Within the MATLAB while loop, the
BufferedReader method readLine reads each line of text (terminated by a
return and/or line feed character) from the site.

    p1 = java.lang.String('<p>');
p2 = java.lang.String('</p>');
s = readLine(br);
while ~(s.startsWith(p1))
    s = readLine(br);
end

5  Display the text.

    disp(s.substring(p1.length,s.length-p2.length))

**Running the Example**

When you run this example, you see output similar to the following.

Explore products for MATLAB, the language of technical computing, and Simulink, for simu

9-65
Find Internet Protocol Address

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Overview” on page 9-66</td>
</tr>
<tr>
<td>“Description of resolveip” on page 9-66</td>
</tr>
<tr>
<td>“Running the Example” on page 9-67</td>
</tr>
</tbody>
</table>

**Overview**

The `resolveip` function returns either the name or address of an IP (internet protocol) host. If you pass `resolveip` a host name, it returns the IP address. If you pass `resolveip` an IP address, it returns the host name. The function uses the `java.net.InetAddress` class, which enables you to find an IP address for a host name, or the host name for a given IP address, without making DNS calls.

`resolveip` calls a static method on the `InetAddress` class to obtain an `InetAddress` object. Then, it calls accessor methods on the `InetAddress` object to get the host name and IP address for the input argument. It displays either the host name or the IP address, depending on the program input argument.

**Description of resolveip**

The major tasks performed by `resolveip` are:

1. Create an `InetAddress` object.

   Instead of constructors, the `java.net.InetAddress` class has static methods that return an instance of the class. The `try` statement calls one of those methods, `getByName`, passing the input argument that the user has passed to `resolveip`. The input argument can be either a host name or an IP address. If `getByName` fails, the `catch` statement displays an error message.

   ```matlab
   function resolveip(input)
   try
       address = java.net.InetAddress.getByName(input);
   catch
       error(sprintf('Unknown host %s.', input))
   end
   ```

2. Retrieve the host name and IP address.
The example uses calls to the `getHostName` and `getHostAddress` accessor functions on the `java.net.InetAddress` object, to obtain the host name and IP address, respectively. These two functions return objects of class `java.lang.String`; use the `char` function to convert them to character arrays.

```java
hostname = char(address.getHostName);
ipaddress = char(address.getHostAddress);
```

Display the host name or IP address.

The example uses the MATLAB `strcmp` function to compare the input argument to the resolved IP address. If it matches, MATLAB displays the host name for the internet address. If the input does not match, MATLAB displays the IP address.

```matlab
if strcmp(input,ipaddress)
    disp(sprintf('Host name of %s is %s', input, hostname))
else
    disp(sprintf('IP address of %s is %s', input, ipaddress))
end
```

**Running the Example**

Here is an example of calling the `resolveip` function with a host name.

```matlab
resolveip ('www.mathworks.com')
IP address of www.mathworks.com is 144.212.100.10
```

Here is a call to the function with an IP address.

```matlab
resolveip ('144.212.100.10')
Host name of 144.212.100.10 is www.mathworks.com
```
Create and Use Phone Book

In this section...

“Overview” on page 9-68
“Description of Function phonebook” on page 9-69
“Description of Function pb_lookup” on page 9-73
“Description of Function pb_add” on page 9-73
“Description of Function pb_remove” on page 9-74
“Description of Function pb_change” on page 9-75
“Description of Function pb_listall” on page 9-76
“Description of Function pb_display” on page 9-77
“Description of Function pb_keyfilter” on page 9-77
“Running the phonebook Program” on page 9-78

Overview

The main function, phonebook, can be called either with no arguments, or with one argument, which is the key of an entry that exists in the phone book. The function first determines the folder to use for the phone book file.

If no phone book file exists, it creates one by constructing a java.io.FileOutputStream object, and then closing the output stream. Next, it creates a data dictionary by constructing an object of java.util.Properties, which is a subclass of java.util.Hashtable, for storing key/value pairs in a hash table. For the phonebook program, the key is a name, and the value is one or more telephone numbers.

The phonebook function creates and opens an input stream for reading by constructing a java.io.FileInputStream object. It calls load on that object to load the hash table contents, if it exists. If the user passed the key to an entry to look up, it looks up the entry by calling pb_lookup, which finds and displays it. Then, the phonebook function returns.

If phonebook was called without the name argument, it then displays a textual menu of the available phone book actions:

• Look up an entry
• Add an entry
• Remove an entry
• Change the phone number(s) in an entry
• List all entries

The menu also has a selection to exit the program. The function uses MATLAB functions to display the menu and to input the user selection.

The `phonebook` function iterates accepting user selections and performing the requested phone book action until the user selects the menu entry to exit. The `phonebook` function then opens an output stream for the file by constructing a `java.io.FileOutputStream` object. It calls `save` on the object to write the current data dictionary to the phone book file. It finally closes the output stream and returns.

**Description of Function `phonebook`**

The major tasks performed by `phonebook` are:

1. Determine the data folder and full file name.

   The first statement assigns the phone book file name, 'myphonebook', to the variable `pbname`. If the `phonebook` program is running on a Windows system, it calls the `java.lang.System.static` method `getProperty` to find the location of the data dictionary. This is set to the user's current working folder. Otherwise, it uses MATLAB function `getenv` to determine the location, using the system variable `HOME`, which you can define beforehand to anything you like. It then assigns to `pbname` the full path, consisting of the data folder and file name 'myphonebook'.

   ```matlab
   function phonebook(varargin)
   pbname = 'myphonebook'; % name of data dictionary
   if ispc
       datadir = char(java.lang.System.getProperty('user.dir'));
   else
       datadir = getenv('HOME');
   end
   pbname = fullfile(datadir, pbname);
   ```

2. If needed, create a file output stream.

   If the `phonebook` file does not exist, `phonebook` asks the user whether to create a new one. If the user answers `y`, `phonebook` creates a phone book by constructing a `FileOutputStream` object. In the `try` clause of a `try/catch` block, the argument `pbname` passed to the `FileOutputStream` constructor is the full name of the file.
that the constructor creates and opens. The next statement closes the file by calling close on the FileOutputStream object FOS. If the output stream constructor fails, the catch statement prints a message and terminates the program.

```matlab
if ~exist(pbname)
    disp(sprintf('Data file %s does not exist.', pbname))
    r = input('Create a new phone book (y/n)?','s');
    if r == 'y',
        try
            FOS = java.io.FileOutputStream(pbname);
            FOS.close
        catch
            error(sprintf('Failed to create %s', pbname))
        end
    else
        return
    end
end
```

3 Create a hash table.

The example constructs a java.util.Properties object to serve as the hash table for the data dictionary.

```matlab
pb_htable = java.util.Properties;
```

4 Create a file input stream.

In a try block, the example invokes a FileInputStream constructor with the name of the phone book file, assigning the object to FIS. If the call fails, the catch statement displays an error message and terminates the program.

```matlab
try
    FIS = java.io.FileInputStream(pbname);
catch
    error(sprintf('Failed to open %s for reading.', pbname))
end
```

5 Load the phone book keys and close the file input stream.

The example calls load on the FileInputStream object FIS, to load the phone book keys and their values (if any) into the hash table. It then closes the file input stream.

```matlab
pb_htable.load(FIS)
FIS.close"
6 Display the Action menu and get the user's selection.

Within a `while` loop, several `disp` statements display a menu of actions that the user can perform on the phone book. Then, an `input` statement requests the user's typed selection.

```
while 1
    disp ' 
    disp ' Phonebook Menu:
    disp ' 1. Look up a phone number'
    disp ' 2. Add an entry to the phone book'
    disp ' 3. Remove an entry from the phone book'
    disp ' 4. Change the contents of an entry in the phone book'
    disp ' 5. Display entire contents of the phone book'
    disp ' 6. Exit this program'
    disp ' 
    s = input('Please type the number for a menu selection: ','s');
```

7 Invoke the function to perform a phone book action

Still within the `while` loop, a `switch` statement provides a case to handle each user selection `s`. Each of the first five cases invokes the function to perform a phone book action.

```
switch s
```

Case 1 prompts for a name that is a key to an entry. It calls `isempty` to determine whether the user has entered a name. If a name has not been entered, it calls `disp` to display an error message. If a name has been input, it passes it to `pb_lookup`. The `pb_lookup` routine looks up the entry and, if it finds it, displays the entry contents.

```
case '1',
    name = input('Enter name to look up: ','s');
    if isempty(name)
        disp 'No name entered'
    else
        pb_lookup(pb_htable, name)
    end
```

Case 2 calls `pb_add`, which prompts the user for a new entry and then adds it to the phone book.
case '2',
    pb_add(pb_htable)

Case 3 uses input to prompt for the name of an entry to remove. If a name has not been entered, it calls disp to display an error message. If a name has been entered, it passes it to pb_remove.

case '3',
    name = input('Enter name of entry to remove: ', 's');
    if isempty(name)
        disp 'No name entered'
    else
        pb_remove(pb_htable, name)
    end

Case 4 uses input to prompt for the name of an entry to change. If a name has not been entered, it calls disp to display an error message. If a name has been entered, it passes it to pb_change.

case '4',
    name = input('Enter name of entry to change: ', 's');
    if isempty(name)
        disp 'No name entered'
    else
        pb_change(pb_htable, name)
    end

Case 5 calls pb_listall to display all entries.

case '5',
    pb_listall(pb_htable)

Exit by creating an output stream and saving the phone book.

8 Exit by creating an output stream and saving the phone book.

If the user has selected case 6 to exit the program, a try statement calls the constructor for a FileOutputStream object, passing it the name of the phone book. If the constructor fails, the catch statement displays an error message.

If the object is created, the next statement saves the phone book data by calling save on the Properties object pb_htable, passing the FileOutputStream object FOS and a descriptive header string. It then calls close on the FileOutputStream object, and returns.
case '6',
    try
        FOS = java.io.FileOutputStream(pbname);
    catch
        error(sprintf('Failed to open %s for writing.',pbname))
    end
    pb_htable.save(FOS,'Data file for phonebook program')
    FOS.close
    return
otherwise
    disp 'That selection is not on the menu.'
end
end % switch

**Description of Function pb_lookup**

Arguments passed to `pb_lookup` are the Properties object `pb_htable` and the name key for the requested entry. The `pb_lookup` function first calls `get` on `pb_htable` with the name key, on which support function `pb_keyfilter` is called to change spaces to underscores. The `get` method returns the entry (or null, if the entry is not found) to variable `entry`. `get` takes an argument of type `java.lang.Object` and also returns an argument of that type. In this invocation, the key passed to `get` and the entry returned from it are actually character arrays.

`pb_lookup` then calls `isempty` to determine whether `entry` is null. If it is, it uses `disp` to display a message stating that the name was not found. If `entry` is not null, it calls `pb_display` to display the entry.

```matlab
function pb_lookup(pb_htable,name)
    entry = pb_htable.get(pb_keyfilter(name));
    if isempty(entry),
        disp(sprintf('The name %s is not in the phone book',name))
    else
        pb_display(entry)
    end
```

**Description of Function pb_add**

1. Input the entry to add.
The `pb_add` function takes one argument, the `Properties` object `pb_htable`. `pb_add` uses `disp` to prompt for an entry. Using the up arrow (^) character as a line delimiter, `input` inputs a name to the variable `entry`. Then, within a `while` loop, it uses `input` to get another line of the entry into variable `line`. If the line is empty, indicating that the user has finished the entry, the code breaks out of the `while` loop. If the line is not empty, the `else` statement appends line to `entry` and then appends the line delimiter. At the end, the `strcmp` checks the possibility that no input was entered and, if that is the case, returns.

```matlab
function pb_add(pb_htable)
    disp 'Type the name for the new entry, followed by Enter.'
    disp 'Then, type the phone number(s), one per line.'
    disp 'To complete the entry, type an extra Enter.'
    name = input(':: ','s');
    entry = [name '^'];
    while 1
        line = input(':: ','s');
        if isempty(line)
            break
        else
            entry = [entry line '^'];
        end
    end
    if strcmp(entry, '^')
        disp 'No name entered'
        return
    end

2 Add the entry to the phone book.

After the input has completed, `pb_add` calls `put` on `pb_htable` with the hash key `name` (on which `pb_keyfilter` is called to change spaces to underscores) and `entry`. It then displays a message that the entry has been added.

```matlab
    pb_htable.put(pb_keyfilter(name),entry)
    disp ''
    disp(sprintf('%s has been added to the phone book.', name))
```

### Description of Function `pb_remove`

1 Look for the key in the phone book.
Arguments passed to `pb_remove` are the Properties object `pb_htable` and the name key for the entry to remove. The `pb_remove` function calls `containsKey` on `pb_htable` with the name key, on which support function `pb_keyfilter` is called to change spaces to underscores. If `name` is not in the phone book, `disp` displays a message and the function returns.

```matlab
function pb_remove(pb_htable,name)
    if ~pb_htable.containsKey(pb_keyfilter(name))
        disp(sprintf('The name %s is not in the phone book',name))
        return
    end

    2 Ask for confirmation and if given, remove the key.

    If the key is in the hash table, `pb_remove` asks for user confirmation. If the user confirms the removal by entering y, `pb_remove` calls `remove` on `pb_htable` with the (filtered) name key, and displays a message that the entry has been removed. If the user enters n, the removal is not performed and `disp` displays a message that the removal has not been performed.

    r = input(sprintf('Remove entry %s (y/n)? ',name), 's');
    if r == 'y'
        pb_htable.remove(pb_keyfilter(name))
        disp(sprintf('%s has been removed from the phone book',name))
    else
        disp(sprintf('%s has not been removed',name))
    end
```

**Description of Function `pb_change`**

1 Find the entry to change, and confirm.

Arguments passed to `pb_change` are the Properties object `pb_htable` and the name key for the requested entry. The `pb_change` function calls `get` on `pb_htable` with the name key, on which `pb_keyfilter` is called to change spaces to underscores. The `get` method returns the entry (or null, if the entry is not found) to variable `entry`. `pb_change` calls `isempty` to determine whether the entry is empty. If the entry is empty, `pb_change` displays a message that the name is added to the phone book, and allows the user to enter the phone number for the entry.

If the entry is found, in the `else` clause, `pb_change` calls `pb_display` to display the entry. It then uses `input` to ask the user to confirm the replacement. If the user enters anything other than y, the function returns.
function pb_change(pb_hetable,name)
entry = pb_hetable.get(pb_keyfilter(name));
if isempty(entry)
    disp(sprintf('The name %s is not in the phone book', name))
    return
else
    pb_display(entry)
    r = input('Replace phone numbers in this entry (y/n)? ','s');
    if r ~= 'y'
        return
    end
end

2 Input new phone numbers and change the phone book entry.

    pb_change uses disp to display a prompt for new phone numbers. Then, 
    pb_change inputs data into variable entry, with the same statements described in 
    “Description of Function pb_lookup” on page 9-73.

Then, to replace the existing entry with the new one, pb_change calls put on 
pb_hetable with the (filtered) key name and the new entry. It then displays a 
message that the entry has been changed.

    disp 'Type in the new phone number(s), one per line.' 
    disp 'To complete the entry, type an extra Enter.' 
    disp(sprintf(':: %s', name))
    entry = [name '^'
    while 1
        line = input(':: ','s');
        if isempty(line)
            break
        else
            entry = [entry line '^'
        end
    end
    pb_hetable.put(pb_keyfilter(name),entry)
    disp ' '
    disp(sprintf('The entry for %s has been changed', name))

Description of Function pb_listall

The pb_listall function takes one argument, the Properties object pb_hetable. 
The function calls propertyNames on the pb_hetable object to return to enum a
java.utilEnumeration object, which supports convenient enumeration of all the keys. In a while loop, pb_listall calls hasMoreElements on enum, and if it returns true, pb_listall calls nextElement on enum to return the next key. It then calls pb_display to display the key and entry, which it retrieves by calling get on pb_htable with the key.

```
function pb_listall(pb_htable)
enum = pb_htable.propertyNames;
while enum.hasMoreElements
    key = enum.nextElement;
    pb_display(pb_htable.get(key))
end
```

**Description of Function pb_display**

The pb_display function takes an argument entry, which is a phone book entry. After displaying a horizontal line, pb_display calls MATLAB function strtok to extract the first line of the entry, up to the line delimiter (\^), into t and the remainder into r. Then, within a while loop that terminates when t is empty, it displays the current line in t. Then it calls strtok to extract the next line from r, into t. When all lines have been displayed, pb_display indicates the end of the entry by displaying another horizontal line.

```
function pb_display(entry)
disp ' ';
disp '-------------------------';
[t,r] = strtok(entry,'^');
while ~isempty(t)
    disp(sprintf(' %s',t))
    [t,r] = strtok(r,'^');
end
disp '-------------------------';
```

**Description of Function pb_keyfilter**

The pb_keyfilter function takes an argument key, which is a name used as a key in the hash table, and either filters it for storage or unfilters it for display. The filter, which replaces each space in the key with an underscore (\_), makes the key usable with the methods of java.util.Properties.

```
function out = pb_keyfilter(key)
if ~isempty(strfind(key,' '))
```
out = strrep(key, ' ', '_');
else
    out = strrep(key, '_', ' ');
end

Running the phonebook Program

In this sample run, a user invokes `phonebook` with no arguments. The user selects menu action 5, which displays the two entries currently in the phone book (all entries are fictitious). Then, the user selects 2, to add an entry. After adding the entry, the user again selects 5, which displays the new entry along with the other two entries.

Phonebook Menu:

1. Look up a phone number
2. Add an entry to the phone book
3. Remove an entry from the phone book
4. Change the contents of an entry in the phone book
5. Display entire contents of the phone book
6. Exit this program

Please type the number for a menu selection: 5

-------------------------
Sylvia Woodland
(508) 111-3456
-------------------------
-------------------------
Russell Reddy
(617) 999-8765
-------------------------

Phonebook Menu:

1. Look up a phone number
2. Add an entry to the phone book
3. Remove an entry from the phone book
4. Change the contents of an entry in the phone book
5. Display entire contents of the phone book
6. Exit this program

Please type the number for a menu selection: 2
Type the name for the new entry, followed by Enter. Then, type the phone number(s), one per line. To complete the entry, type an extra Enter.

:: BriteLites Books
:: (781) 777-6868
::

BriteLites Books has been added to the phone book.

Phonebook Menu:

1. Look up a phone number
2. Add an entry to the phone book
3. Remove an entry from the phone book
4. Change the contents of an entry in the phone book
5. Display entire contents of the phone book
6. Exit this program

Please type the number for a menu selection: 5

-------------------------
BriteLites Books
(781) 777-6868
-------------------------
-------------------------
Sylvia Woodland
(508) 111-3456
-------------------------
-------------------------
Russell Reddy
(617) 999-8765
-------------------------
Java Heap Memory Preferences

You can adjust the amount of memory that MATLAB software 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, 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, check your Java code for memory leaks. Eliminate references to objects that are no longer useful. For more information, see the Java SE Troubleshooting guide at [http://www.oracle.com/technetwork/java/javase/index-138283.html](http://www.oracle.com/technetwork/java/javase/index-138283.html).
Using .NET Libraries from MATLAB

- “Read Cell Arrays of Excel Spreadsheet Data” on page 10-4
- “Access a Simple .NET Class” on page 10-6
- “Load a Global .NET Assembly” on page 10-11
- “Work with Microsoft Excel Spreadsheets Using .NET” on page 10-12
- “Work with Microsoft Word Documents Using .NET” on page 10-14
- “Assembly is Library of .NET Classes” on page 10-15
- “Limitations to .NET Support” on page 10-16
- “System Requirements for Using MATLAB Interface to .NET” on page 10-18
- “Using .NET from MATLAB” on page 10-19
- “Using a .NET Object” on page 10-21
- “Build a .NET Application for MATLAB Examples” on page 10-23
- “Troubleshooting Security Policy Settings From Network Drives” on page 10-24
- “.NET Terminology” on page 10-25
- “Simplify .NET Class Names” on page 10-26
- “Use import in MATLAB Functions” on page 10-27
- “Nested Classes” on page 10-28
- “Handle .NET Exceptions” on page 10-29
- “Pass Numeric Arguments” on page 10-30
- “Pass System.String Arguments” on page 10-31
- “Pass System.Enum Arguments” on page 10-33
- “Pass System.Nullable Arguments” on page 10-35
- “Pass Cell Arrays of .NET Data” on page 10-39
- “Pass Jagged Arrays” on page 10-42
- “Convert Nested System.Object Arrays” on page 10-45
- “Pass Data to .NET Objects” on page 10-46
• “Handle Data Returned from .NET Objects” on page 10-53
• “Use Arrays with .NET Applications” on page 10-58
• “Convert .NET Arrays to Cell Arrays” on page 10-60
• “Limitations to Support of .NET Arrays” on page 10-63
• “Set Static .NET Properties” on page 10-64
• “Using .NET Properties” on page 10-66
• “MATLAB Does Not Display Protected Properties” on page 10-68
• “Work with .NET Methods Having Multiple Signatures” on page 10-69
• “Call .NET Methods With out Keyword” on page 10-71
• “Call .NET Methods With ref Keyword” on page 10-73
• “Call .NET Methods With params Keyword” on page 10-75
• “Call .NET Methods with Optional Arguments” on page 10-77
• “Calling .NET Methods” on page 10-80
• “Calling .NET Methods with Optional Arguments” on page 10-83
• “Calling .NET Extension Methods” on page 10-84
• “Call .NET Properties That Take an Argument” on page 10-85
• “How MATLAB Represents .NET Operators” on page 10-87
• “Limitations to Support of .NET Methods” on page 10-88
• “Use .NET Events in MATLAB” on page 10-89
• “Call .NET Delegates in MATLAB” on page 10-92
• “Create Delegates from .NET Object Methods” on page 10-94
• “Create Delegate Instances Bound to .NET Methods” on page 10-95
• “Call Delegates With out and ref Type Arguments” on page 10-97
• “Combine and Remove .NET Delegates” on page 10-98
• “.NET Delegates” on page 10-100
• “Calling .NET Methods Asynchronously” on page 10-101
• “Limitations to Support of .NET Events” on page 10-105
• “Limitations to Support of .NET Delegates” on page 10-106
• “Use Bit Flags with .NET Enumerations” on page 10-107
• “Read Special System Folder Path” on page 10-111
Using .NET Libraries from MATLAB

- “.NET Enumerations in MATLAB” on page 10-112
- “Default Methods for an Enumeration” on page 10-113
- “NetDocEnum Example Assembly” on page 10-115
- “Work with Members of a .NET Enumeration” on page 10-116
- “Refer to a .NET Enumeration Member” on page 10-118
- “Display .NET Enumeration Members as Character Strings” on page 10-120
- “Convert .NET Enumeration Values to Type Double” on page 10-121
- “Iterate Through a .NET Enumeration” on page 10-122
- “Use .NET Enumerations to Test for Conditions” on page 10-124
- “Underlying Enumeration Values” on page 10-126
- “Limitations to Support of .NET Enumerations” on page 10-127
- “Create .NET Collections” on page 10-128
- “Convert .NET Collections to MATLAB Arrays” on page 10-130
- “Create .NET Arrays of Generic Type” on page 10-131
- “Display .NET Generic Methods Using Reflection” on page 10-132
- “.NET Generic Classes” on page 10-135
- “Accessing Items in .NET Collections” on page 10-136
- “Call .NET Generic Methods” on page 10-137
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 strings and System.DateTime values to datetime objects.

Create a file in Excel that contains the following data.

<table>
<thead>
<tr>
<th>Date</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/31/96</td>
<td>174.8</td>
</tr>
<tr>
<td>11/29/96</td>
<td>179.3</td>
</tr>
<tr>
<td>12/30/96</td>
<td>190.4</td>
</tr>
<tr>
<td>01/31/97</td>
<td>185.7</td>
</tr>
</tbody>
</table>

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.

```matlab
NET.addAssembly('microsoft.office.interop.excel');
app = Microsoft.Office.Interop.Excel.ApplicationClass;
book = app.Workbooks.Open('H:\Documents\MATLAB\weight.xls');
range = sheet.UsedRange;
arr = range.Value;
```

Convert the data to MATLAB types.

```matlab
data = cell(arr,'ConvertTypes',{'all'});
```

Display the dates.

```matlab
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)

**Related Examples**
- “Read Spreadsheet Data Using Excel as Automation Server” on page 11-23

**More About**
- “Convert .NET Arrays to Cell Arrays” on page 10-60
- “Spreadsheets”
Access a Simple .NET Class

In this section...

“System.DateTime Example” on page 10-6
“Create .NET Object From Constructor” on page 10-7
“View Information About .NET Object” on page 10-7
“Introduction to .NET Data Types” on page 10-9

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 10-20.
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`.

```matlab
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.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>System.DateTime obj</code></td>
<td><code>DateTime</code></td>
<td><code>(int32 scalar year, etc.)</code></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>System.DateTime obj</code></td>
<td><code>DateTime</code></td>
<td><code>(int32 scalar year, int32 scalar month, int32 scalar day)</code></td>
</tr>
</tbody>
</table>

For more information, see “Reading Method Signatures” on page 10-81.

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:

```matlab
% 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**

```matlab
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:

```matlab
    Date
    Day
    DayOfWeek
    DayOfYear
    Hour
    Kind
    Millisecond
    Minute
    Month
    Now
    UtcNow
```
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 isValid
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 10-66
• “Calling .NET Methods” on page 10-80

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 10-53 and “Numeric Types”.

- Strings (System.String) — Use the char function to convert a System.String object to a MATLAB string:

  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 “Enumerations”.

For a complete list of supported types and mappings, see “Handle Data Returned from .NET Objects” on page 10-53.
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:

```matlab
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.

```matlab
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 10-6 does not call NET.addAssembly because MATLAB dynamically loads its assembly (mscorlib) at startup.

---

**Note:** You cannot unload an assembly in MATLAB.

For more information, see:

- “Assembly is Library of .NET Classes” on page 10-15
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 http://msdn.microsoft.com/.

To create a workbook, type:

```matlab
NET.addAssembly('microsoft.office.interop.excel');
app = Microsoft.Office.Interop.Excel.ApplicationClass;
books = app.Workbooks;
newWB = Add(books);
app.Visible = true;
```

Create a new sheet:

```matlab
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:

```matlab
newWS = Microsoft.Office.Interop.Excel.Worksheet(newSheet);
```

Create MATLAB data and write columns 1 and 2 to a range of cells.

```matlab
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.

```matlab
% 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:

```csharp
newFont = newRange.Font;
newFont.Bold = 1;
newWS.Name = 'Test Data';
```

If this is a new spreadsheet, use the `SaveAs` method:

```csharp
SaveAs(newWB,'mySpreadsheet.xlsx');
```

Close and quit:

```csharp
Close(newWB)
Quit(app)
```
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 http://msdn.microsoft.com/.

The following code creates a new Word document:

```matlab
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:

```matlab
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:

```matlab
SaveAs(newDoc,'myDocument.docx');
```

When you are finished, to close the document and application, type:

```matlab
Save(newDoc);
Close(newDoc);
Quit(wordApp);
```
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 directory, 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.

<table>
<thead>
<tr>
<th>Features Not Supported in MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannot use ClassName.propertyname syntax to set static properties. Use NET.setStaticProperty instead.</td>
</tr>
<tr>
<td>Unloading an assembly</td>
</tr>
<tr>
<td>Passing a structure array, sparse array, or complex number to a .NET property or method</td>
</tr>
<tr>
<td>Subclassing .NET classes from MATLAB</td>
</tr>
<tr>
<td>Accessing nonpublic class members</td>
</tr>
<tr>
<td>Displaying generic methods using methods or methodsview functions. For a workaround, see “Display .NET Generic Methods Using Reflection” on page 10-132.</td>
</tr>
<tr>
<td>Creating an instance of a nested class. For a workaround, see “Nested Classes” on page 10-28.</td>
</tr>
<tr>
<td>Saving (serializing) .NET objects into a MAT-file</td>
</tr>
<tr>
<td>Creating .NET arrays with a specific lower bound</td>
</tr>
<tr>
<td>Concatenating multiple .NET objects into an array</td>
</tr>
<tr>
<td>Implementing interface methods</td>
</tr>
<tr>
<td>Hosting .NET controls in figure windows</td>
</tr>
<tr>
<td>Casting operations</td>
</tr>
<tr>
<td>Calling constructors with ref or out type arguments</td>
</tr>
<tr>
<td>Using System.Console.WriteLine to write text to the command window</td>
</tr>
<tr>
<td>Pointer type arguments, function pointers, Dllimport keyword</td>
</tr>
<tr>
<td>.NET remoting</td>
</tr>
<tr>
<td>Using the MATLAB : (colon) operator in a foreach iteration</td>
</tr>
<tr>
<td>Adding event listeners to .NET events defined in static classes</td>
</tr>
<tr>
<td>Handling .NET events with signatures that do not conform to the standard signature</td>
</tr>
<tr>
<td>Creating empty .NET objects</td>
</tr>
<tr>
<td>Features Not Supported in MATLAB</td>
</tr>
<tr>
<td>----------------------------------</td>
</tr>
<tr>
<td>Creating .NET objects that do not belong to a namespace</td>
</tr>
</tbody>
</table>
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 vendor’s 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/arch 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, please visit the Configuration File Schema for the .NET Framework Web page at http://msdn.microsoft.com/en-us/library/1fk1t1t0.aspx.
Using .NET from MATLAB

In this section...

“Benefits of the MATLAB .NET Interface” on page 10-19
“Why Use the MATLAB .NET Interface?” on page 10-19
“NET Assembly Integration Using MATLAB Compiler SDK” on page 10-20
“To Learn More About the .NET Framework” on page 10-20

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:

```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 at [http://msdn.microsoft.com/en-us/netframework/aa496123](http://msdn.microsoft.com/en-us/netframework/aa496123) 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 10-18 for information about version support in MATLAB.

**More About**

- “Limitations to .NET Support” on page 10-16
Using a .NET Object

In this section...

“Creating a .NET Object” on page 10-21
“What Classes Are in a .NET Assembly?” on page 10-21
“Using the delete Function on a .NET Object” on page 10-22

Creating a .NET Object

You often need to 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 more information about objects, see “MATLAB 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 10-7.

To call method methodName:

returnedValue = methodName(classObj,args,...)

For more information, see:

• “Properties”
• “Methods”
• “Events and Delegates”

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 `mscorlib` library, type:

```matlab
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 10-20.

**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, see “Destroying Objects”.
Build a .NET Application for MATLAB Examples

You can use C# code examples in MATLAB, such as the NetDocCell11 assembly provided in “Convert .NET Arrays to Cell Arrays” on page 10-60. 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 to do this; 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.
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.m file, from the matlabroot\toolbox\matlab\winfun\NET folder.

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 make changes to 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’s data is also copied and the new object is independent of changes to the original object. For more information about these differences, see “Copying Objects”.

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 10-6 example, type:

```matlab
import System.*
import System.DateTime.*
```

To create the object, type:

```matlab
netDate = DateTime.Today;
```

To use a static method, type:

```matlab
DaysInMonth(netDate.Year, netDate.Month)
```
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.

```matlab
function ptr = getPrinterInfo
import System.Drawing.Printing.*;
ptr = PrinterSettings;
end
```

To call the function, type:

```matlab
NET.addAssembly('System.Drawing');
printer = getPrinterInfo;
```

You cannot 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:

```matlab
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:

```csharp
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:

```matlab
a = NET.addAssembly('C:\Work\MyClassLibrary.dll');
a.Classes
ans =
'MyClassLibrary.OuterClass'
'MyClassLibrary.OuterClass+InnerClass'
```

To call strmethod, type:

```matlab
t = a.AssemblyHandle.GetType('MyClassLibrary.OuterClass+InnerClass');
sa = System.Activator.CreateInstance(t);
strmethod(sa,'hello')
ans =
from InnerClass hello
```
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
```
Pass Numeric Arguments

In this section...

“Call .NET Methods with Numeric Arguments” on page 10-30
“Use .NET Numeric Types in MATLAB” on page 10-30

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 10-46.

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 10-53.

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 10-31
“Use System.String in MATLAB” on page 10-31

Call .NET Methods with System.String Arguments

If an input argument to a .NET method is System.String, you can pass a MATLAB string. MATLAB automatically converts a char array (string) argument into System.String. For example, the following code uses the System.DateTime.Parse method to convert a date represented by a string 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.

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>System.DateTime</td>
<td>Parse</td>
<td>(System.String s)</td>
</tr>
<tr>
<td></td>
<td>RetVal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more information, see:

• “Pass MATLAB Strings” on page 10-48

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.
netDate = System.DateTime.Now;
thisTime = ToShortTimeString(netDate);
class(thisTime)

ans =
System.String

The current time, thisTime, is a System.String object.

To display thisTime in MATLAB, use the char function to convert the System.String object to a MATLAB string.

disp(['The time is ' char(thisTime)])

More About
• “How MATLAB Handles System.String” on page 10-54
Pass System.Enum Arguments

In this section...

“Call .NET Methods with System.Enum Arguments” on page 10-33
“Use System.Enum in MATLAB” on page 10-34

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”.

```matlab
% 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:

```matlab
methodsview('System.Globalization.DateTimeFormatInfo')
```

Search the list for GetAbbreviatedDayName.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.String RetVal</td>
<td>GetAbbreviatedDayName</td>
<td>( System.Globalization.DateTimeFormatInfo this, System.DayOfWeek dayofweek)</td>
</tr>
</tbody>
</table>

For more information, see:

• “.NET Enumerations in MATLAB” on page 10-112
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:

```matlab
myDay = System.DateTime.Today;
disp(myDay.DayOfWeek)
```

- Use an enumeration in comparison statements. For example:

```matlab
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:

```matlab
myDay = System.DateTime.Today;
dow = myDay.DayOfWeek;
startDateOfWeek = AddDays(myDay,-double(dow));
ToShortDateString(startDateOfWeek)
```

- Perform bit-wise operations. For examples, see “Creating .NET Enumeration Bit Flags” on page 10-107.

For more information, see:

- “Iterate Through a .NET Enumeration” on page 10-122
- “Use .NET Enumerations to Test for Conditions” on page 10-124
- “Use Bit Flags with .NET Enumerations” on page 10-107
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 how 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

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
</table>

Load NetDocNullable Assembly

The example assumes 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.

```matlab
field1 = SetField(cls,10)
```

```matlab
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, `[]`.

```matlab
field2 = SetField(cls,[])
```

```matlab
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.

```matlab
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

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double scalar RetVal</td>
<td>GetValueOrDefault</td>
<td>(System.Nullable &lt;System*Double&gt; this)</td>
</tr>
</tbody>
</table>

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 =
Call `plot`:

```matlab
plot([0 defaultData])
```

You can change the default value using the `GetValueOrDefault` method. Open the `methodsview` window and look for the following function signature:

**GetValueOrDefault Function Signature to Change Default**

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double scalar RetVal</td>
<td>GetValueOrDefault</td>
<td>(System.Nullable&lt;System*Double&gt; this, double scalar defaultValue)</td>
</tr>
</tbody>
</table>

Set the `defaultValue` input argument to a new value, -1, and plot the results for null value `field2`.

```matlab
defaultData = GetValueOrDefault(field2,-1);
plot([0 defaultData])
```

**Related Examples**

- “Build a .NET Application for MATLAB Examples” on page 10-23

**More About**

- “Pass System.Nullable Type” on page 10-48
- “How MATLAB Handles System.Nullable” on page 10-56

**External Websites**

Pass Cell Arrays of .NET Data

In this section...

“Example of Cell Arrays of .NET Data” on page 10-39
“Create a Cell Array for Each System.Object” on page 10-40
“Create MATLAB Variables from the .NET Data” on page 10-40
“Call MATLAB Functions with MATLAB Variables” on page 10-40

Example of Cell Arrays of .NET Data

In the “Convert Nested System.Object Arrays” on page 10-60 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:

```matlab
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 10-53.
- 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:

```matlab
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:

```matlab
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:

```matlab
% 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:

```matlab
% Remove the previous figure
contre
% 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 10-42
“Call .NET Method with System.String Jagged Array Arguments” on page 10-42
“Call .NET Method with Multidimensional Jagged Array Arguments” on page 10-43

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 3 element array.

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

You can pass jArr to any .NET method with an input or output argument of type System.Double[].

Call .NET Method with System.String Jagged Array Arguments

This example shows how to create an array of MATLAB strings to pass to a method, MethodStringArr, with a System.String[][] input argument.

The following is the MATLAB function signature for MethodStringArr.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.String[][]</td>
<td>MethodStringArr</td>
<td>(NetPackage.StringClass this,</td>
</tr>
</tbody>
</table>
The MATLAB strings you want to pass to the method are:

```plaintext
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.

```plaintext
netArr = NET.createArray('System.String[]',2);
```

The arrays contain empty strings.

Create `System.String` arrays to correspond to the MATLAB strings, `str1` and `str2`.

```plaintext
netArr(1) = NET.createArray('System.String',2);
netArr(2) = NET.createArray('System.String',1);
```

Assign `str1` and `str2` to `netArr`.

```plaintext
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.

```plaintext
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.
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.

```matlab
dllPath = fullfile('c:','work','NetDocCell.dll');
NET.addAssembly(dllPath);
```

Create a cell array, `mlData`.

```matlab
test = NetDocCell.MyGraph;
mldata = cell(test.getNewData)
mlData = 
    [1x1 System.String]    [1x1 System.Object[]]
```

To access the contents of the `System.Object` array, create another cell array `mlPlotData`.

```matlab
mlPlotData = cell(mlData{2})
mlPlotData =
    [1x1 System.String]    [1x1 System.Double[]]
```

Related Examples

- “Pass Cell Arrays of .NET Data” on page 10-39

More About

- “Build a .NET Application for MATLAB Examples” on page 10-23
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 to .NET.

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Pass Primitive .NET Types” on page 10-46</td>
</tr>
<tr>
<td>“Pass Cell Arrays” on page 10-47</td>
</tr>
<tr>
<td>“Pass Nonprimitive .NET Objects” on page 10-48</td>
</tr>
<tr>
<td>“Pass MATLAB Strings” on page 10-48</td>
</tr>
<tr>
<td>“Pass System.Nullable Type” on page 10-48</td>
</tr>
<tr>
<td>“Pass NULL Values” on page 10-49</td>
</tr>
<tr>
<td>“Unsupported MATLAB Types” on page 10-49</td>
</tr>
<tr>
<td>“Choosing Method Signatures” on page 10-49</td>
</tr>
<tr>
<td>“Example — Choosing a Method Signature” on page 10-50</td>
</tr>
<tr>
<td>“Pass Arrays” on page 10-51</td>
</tr>
<tr>
<td>“Pass MATLAB Arrays as Jagged Arrays” on page 10-52</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MATLAB Type</th>
<th>Closest Type</th>
<th>Other Matching .NET Types</th>
<th>Least Close Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>Boolean</td>
<td>SByte</td>
<td>Int32</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>Decim</td>
<td>Int64</td>
</tr>
<tr>
<td>single</td>
<td>Single</td>
<td>Decim</td>
<td>Object</td>
</tr>
</tbody>
</table>
### MATLAB Type Conversion Table

<table>
<thead>
<tr>
<th>MATLAB Type</th>
<th>Closest Type &lt;—— Other Matching .NET Types ———&gt; Least Close Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>SByte, Int16, Int32, Int64, Single, Double, Object</td>
</tr>
<tr>
<td>uint8</td>
<td>Byte, UInt16, UInt32, UInt64, Single, Double, Object</td>
</tr>
<tr>
<td>int16</td>
<td>Int16, Int32, Int64, Single, Double, Object</td>
</tr>
<tr>
<td>uint16</td>
<td>UInt16, UInt32, UInt64, Single, Double, Object</td>
</tr>
<tr>
<td>int32</td>
<td>Int32, Int64, Single, Double, Object</td>
</tr>
<tr>
<td>uint32</td>
<td>UInt32, UInt64, Single, Double, Object</td>
</tr>
<tr>
<td>int64</td>
<td>Int64, Double, Object</td>
</tr>
<tr>
<td>uint64</td>
<td>UInt64, Double, Object</td>
</tr>
<tr>
<td>char</td>
<td>Char, String, Object</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MATLAB Type</th>
<th>Closest Type &lt;—— Other Matching .NET Types ———&gt; Least Close Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array (not all strings)</td>
<td>System.Object[ ]</td>
</tr>
</tbody>
</table>
• Any non-sparse, non-complex built-in numeric type shown in the MATLAB Primitive Type Conversion Table
• 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 Strings

MATLAB automatically converts a string or char array to a .NET System.String object. To pass an array of strings, create a cell array.

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 10-35.
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 10-83.

**Example — Choosing a Method Signature**

Open a `methodsview` window for the `System.String` class and look at the entries for the `Concat` method:

```matlab
import System.*
methodsview('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:

```matlab
str1 = String('hello');
str2 = String('world');
```

When you type:

```matlab
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.

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td><code>System.String RetVal</code></td>
<td><code>Concat</code></td>
<td><code>(System.Object arg0, System.Object arg1)</code></td>
</tr>
<tr>
<td>Static</td>
<td><code>System.String RetVal</code></td>
<td><code>Concat</code></td>
<td><code>(System.String str0, System.String str1)</code></td>
</tr>
</tbody>
</table>

Since `str1` and `str2` are of class `System.String`, MATLAB chooses the second signature and displays:

```matlab
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:

```matlab
dt = DateTime.Today;
myDate = System.DateTime(dt.Year,3,1,11,32,5);
```
When you type:

```matlab
String.Concat(dt,myDate)
```

MATLAB chooses the following signature, since `System.DateTime` objects are derived from the `System.Object` class.

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td><code>System.String</code></td>
<td>Concat</td>
<td><code>(System.Object arg0, System.Object arg1)</code></td>
</tr>
<tr>
<td></td>
<td><code>RetVal</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This `Concat` method first applies the `ToString` method to the objects, then concatenates the strings. MATLAB displays information like:

```matlab
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 10-58 and “Pass MATLAB Arrays as Jagged Arrays” on page 10-52.

**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 non-numeric 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 10-42.
Handle Data Returned from .NET Objects

In this section...

“.NET Type to MATLAB Type Mapping” on page 10-53
“How MATLAB Handles System.String” on page 10-54
“How MATLAB Handles System.__ComObject” on page 10-55
“How MATLAB Handles System.Nullable” on page 10-56
“How MATLAB Handles dynamic Type” on page 10-57
“How MATLAB Handles Jagged Arrays” on page 10-57

.NET Type to MATLAB Type Mapping

The following table shows how MATLAB converts data from a .NET object into MATLAB types. These values are displayed in a method signature.

<table>
<thead>
<tr>
<th>C# .NET Type</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Int16</td>
<td>int16 scalar</td>
</tr>
<tr>
<td>System.UInt16</td>
<td>uint16 scalar</td>
</tr>
<tr>
<td>System.Int32</td>
<td>int32 scalar</td>
</tr>
<tr>
<td>System.UInt32</td>
<td>uint32 scalar</td>
</tr>
<tr>
<td>System.Int64</td>
<td>int64 scalar</td>
</tr>
<tr>
<td>System.UInt64</td>
<td>uint64 scalar</td>
</tr>
<tr>
<td>System.Single</td>
<td>single scalar</td>
</tr>
<tr>
<td>System.Double</td>
<td>double scalar</td>
</tr>
<tr>
<td>System.Boolean</td>
<td>logical scalar</td>
</tr>
<tr>
<td>System.Byte</td>
<td>uint8 scalar</td>
</tr>
<tr>
<td>System.Enum</td>
<td>enum</td>
</tr>
<tr>
<td>System.Char</td>
<td>char</td>
</tr>
<tr>
<td>System.Decimal</td>
<td>System.Decimal</td>
</tr>
<tr>
<td>System.Object</td>
<td>System.Object</td>
</tr>
</tbody>
</table>
### How MATLAB Handles System.String

Use the `char` function to convert a `System.String` object to a MATLAB string. For example, type:

```matlab
str = System.String('create a System.String');
strml = char(str);
whos
```

MATLAB displays the string value of `System.String` objects, instead of the standard object display. For example, type:

```matlab
a = System.String('test')
b = String.Concat(a,' hello',' world')
```

MATLAB displays:
```
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 10-85. 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 10-20.

**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.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NetDocTest.TestType</code></td>
<td><code>GetTestType</code></td>
<td><code>(NetDocTest.MyClass this)</code></td>
</tr>
</tbody>
</table>

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:

```matlab
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:

\[
\text{objConverted} = \text{namespace.classname(obj)}
\]

where \text{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 \text{mySheet} to a Chart or a Worksheet object \text{newSheet}:

\[
\text{newSheet} = \text{Microsoft.Office.Interop.Excel.interfacename(mySheet)};
\]

where \text{interfacename} is Chart or Worksheet. For an example, see “Work with Microsoft Excel Spreadsheets Using .NET” on page 10-12.

**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 \text{NET.disableAutoRelease} function. Then unlock the object, using the \text{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 <Value> values, use the HasValue property.
- If you want every value to be of the underlying <Value>, use the GetValueOrDefault method. This method assigns a default value of type <Value> 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 10-35.
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:

```csharp
public dynamic exampleMethod(dynamic d)
```

The following table shows the corresponding MATLAB function signature.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>System.Object</code></td>
<td><code>RetVal</code></td>
<td><code>(namespace.classname this, System.Object d)</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Passing MATLAB Arrays to .NET” on page 10-58</td>
</tr>
<tr>
<td>“Accessing .NET Array Elements in MATLAB” on page 10-58</td>
</tr>
<tr>
<td>“Converting .NET Jagged Arrays to MATLAB Arrays” on page 10-59</td>
</tr>
</tbody>
</table>

Passing 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 strings, 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#.

Accessing .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 “Matrix 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 “Generating a Numeric Sequence”, 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:

```matlab
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.

**Converting .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**

`cell` | `NET.createArray`

**More About**

- “Convert .NET Arrays to Cell Arrays” on page 10-60
- “Limitations to Support of .NET Arrays” on page 10-63
Convert .NET Arrays to Cell Arrays

In this section...

“Convert Nested System.Object Arrays” on page 10-60
“cell Function Syntax for System.Object[,] Arrays” on page 10-61

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 10-53.

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 10-23. 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 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 strings:

- {'System.DateTime'} — Convert System.DateTime elements to MATLAB datetime elements.
- {'all'} — Convert all supported .NET types to equivalent MATLAB types.

A is a cell array, that is the same size as the obj array.

**Related Examples**

- “Pass Cell Arrays of .NET Data” on page 10-39
- “Read Cell Arrays of Excel Spreadsheet Data” on page 10-4
Limitations to Support of .NET Arrays

MATLAB does not support:

- Arrays which specify a lower bound
- Concatenating .NET objects into an array
- The \texttt{end} function as the last index in a .NET array
- Array indices of complex values
- Autoconversion of \texttt{char} or cell arrays to jagged array arguments.
- Autoconversion of MATLAB arrays to mulidimensional jagged array arguments.
Set Static .NET Properties

In this section...

“Do Not Use ClassName.PropertyName Syntax for Static Properties” on page 10-64


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 to a specific path, for example:

```matlab
cd('C:\Work')
```

Set the CurrentDirectory property:

```matlab
newDir = [char(saveDir) '\temp\'];
mkdir(newDir)
```

```matlab
ans = 
C:\Work\temp
```

To restore the original CurrentDirectory value, type:

```matlab
```

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:

```csharp
newDir = [char(saveDir) '\temp'];
whos
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>1x1</td>
<td>376</td>
<td>struct</td>
</tr>
<tr>
<td>newDir</td>
<td>1x12</td>
<td>24</td>
<td>char</td>
</tr>
<tr>
<td>saveDir</td>
<td>1x1</td>
<td>112</td>
<td>System.String</td>
</tr>
</tbody>
</table>

Try to use a member of the System namespace.

```csharp
oldDate = System.DateTime(1992,3,1);
```

Reference to non-existent field 'DateTime'.

To restore your environment, type:

```csharp
clear System
```
Using .NET Properties

In this section...

“How MATLAB Represents .NET Properties” on page 10-66
“How MATLAB Maps C# Property and Field Access Modifiers” on page 10-66

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:

```matlab
x = ClassName.PropertyName;
ClassName.PropertyName = y;
```

The following example gets the value of a property (the current day of the month):

```matlab
dtnow = System.DateTime.Now;
d = dtnow.Day;
```

The following example sets the value of a property (the `Volume` for a `SpeechSynthesizer` object):

```matlab
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 10-64.

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 10-85.

How MATLAB Maps C# Property and Field Access Modifiers

MATLAB maps C# keywords to MATLAB property attributes, as shown in the following table.
MATLAB maps C# keywords to MATLAB field attributes, as shown in the following table.

<table>
<thead>
<tr>
<th>C# Property Keyword</th>
<th>MATLAB Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>public, static</td>
<td>Access = public</td>
</tr>
<tr>
<td>protected, private, internal</td>
<td>Not visible to MATLAB</td>
</tr>
<tr>
<td>get, set</td>
<td>Access = public</td>
</tr>
<tr>
<td>Get</td>
<td>GetAccess = public, SetAccess = private</td>
</tr>
<tr>
<td>Set</td>
<td>SetAccess = public, GetAccess = private</td>
</tr>
</tbody>
</table>

For more information about MATLAB properties, see “Property Attributes”.

MATLAB maps C# keywords to MATLAB field attributes, as shown in the following table.

<table>
<thead>
<tr>
<th>C# Field Keyword</th>
<th>MATLAB Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Supported</td>
</tr>
<tr>
<td>protected, private, internal, protected internal</td>
<td>Not visible to MATLAB</td>
</tr>
</tbody>
</table>
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');
```

Display Public Properties


- 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 10-66.
Work with .NET Methods Having Multiple Signatures

To create the NetSample assembly, see “Build a .NET Application for MATLAB Examples” on page 10-23.

The SampleMethodSignature class defines the three constructors shown in the following table.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>netdoc.SampleMethodSignature obj</td>
<td>SampleMethodSignature</td>
<td></td>
</tr>
<tr>
<td>netdoc.SampleMethodSignature obj</td>
<td>SampleMethodSignature</td>
<td>(double scalar d)</td>
</tr>
<tr>
<td>netdoc.SampleMethodSignature obj</td>
<td>SampleMethodSignature</td>
<td>(System.String s)</td>
</tr>
</tbody>
</table>

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:

```matlab
NET.addAssembly('c:\work\NetSample.dll')
```

Create a **SampleMethodSignature** object `obj`:

```matlab
obj = netdoc.SampleMethodSignature;
```

To see the method signatures, type:

```matlab
methods(obj,'-full')
```

Look for the following signatures in the MATLAB output:

```matlab
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 10-53.
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.

```csharp
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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double scalar db2</td>
<td>outTest</td>
<td>(netdoc.SampleOutTest this, double scalar db1)</td>
</tr>
</tbody>
</table>

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:

```matlab
asmpath = 'c:\work\Visual Studio 2012\Projects\SampleOutTest\SampleOutTest\bin\Debug\';
asmname = 'SampleOutTest.dll';
```

Load the assembly.

```matlab
asm = NET.addAssembly(fullfile(asmpath,asmname));
```

Call the method.

```matlab
cls = netdoc.SampleOutTest;
```
db3 = outTest(cls,6)

db3 =
   14.1000

**Related Examples**

- “Build a .NET Application for MATLAB Examples” on page 10-23

**More About**

- “C# Method Access Modifiers” on page 10-80
This example shows how to call methods that use a `ref` keyword in the argument list.

The input argument `db1` in the following `refTest` method is modified by the `ref` keyword.

```csharp
using System;
namespace netdoc
{
    public class SampleRefTest
    {
        //test ref keyword
        public void refTest(ref double db1)
        {
            db1 = db1 * 2;
        }
    }
}
```

The function signature in MATLAB is:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double scalar db1</td>
<td>refTest</td>
<td>(netdoc.SampleRefTest this, double scalar db1)</td>
</tr>
</tbody>
</table>

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:

```matlab
asmpath = 'c:\work\Visual Studio 2012\Projects\SampleRefTest\SampleRefTest\bin\Debug\';
asmname = 'SampleRefTest.dll';
```

Load the assembly.

```matlab
asm = NET.addAssembly(fullfile(asmpath,asmname));
```

Call the method.

```matlab
cls = netdoc.SampleRefTest;
db4 = refTest(cls,6)
```
db4 =
12

Related Examples
• “Build a .NET Application for MATLAB Examples” on page 10-23

More About
• “C# Method Access Modifiers” on page 10-80
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.

```csharp
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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 scalar</td>
<td>RetVal</td>
<td>paramsTest (netdoc.SampleParamsTest this, System.Int32[] num)</td>
</tr>
</tbody>
</table>

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:

```matlab
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

Related Examples
• “Build a .NET Application for MATLAB Examples” on page 10-23

More About
• “C# Method Access Modifiers” on page 10-80
Call .NET Methods with Optional Arguments

In this section...

“Setting Up the Examples” on page 10-77
“Skip Optional Arguments” on page 10-77
“Call Overloaded Methods” on page 10-78

Setting Up the Examples

To use the examples in this topic, build the NetDocOptional assembly. This C# example, NetDocOptional.cs in the matlabroot/extern/examples/NET/NetSample folder, defines the methods used in these examples. To see the code, open the file in MATLAB Editor. To build the NetDocOptional assembly, see “Build a .NET Application for MATLAB Examples” on page 10-23. The examples assume you put the assembly in your c:\work folder. You can modify the examples to change the path to the assembly.

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.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
</table>

Load the NetDocOptional assembly, if it is not already loaded.

dllPath = fullfile('c:','work','NetDocOptional.dll');
asm = NET.addAssembly(dllPath);
cls = NetDocOptional.MyClass;
Display the default values.

Greeting(cls,0)

ans =
hello world

Use the default value for str1.

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`.

**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`.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>single scalar RetVal</td>
<td>calc</td>
<td>(NetDocOptional.MyClass this, optional&lt;int32 scalar&gt; x, optional/single scalar&gt; y)</td>
</tr>
<tr>
<td>double scalar RetVal</td>
<td>calc</td>
<td>(NetDocOptional.MyClass this, optional&lt;int32 scalar&gt; x, optional&lt;double scalar&gt; y)</td>
</tr>
</tbody>
</table>

Load the NetDocOptional assembly, if it is not already loaded.

dllPath = fullfile('c:\work', 'NetDocOptional.dll');
asm = NET.addAssembly(dllPath);
cls = NetDocOptional.MyClass;

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':

'NetDocOptional.MyClass.calc(NetDocOptional.MyClass this, optional<int32 scalar> x, optional<single scalar> y)' and
'NetDocOptional.MyClass.calc(NetDocOptional.MyClass 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.

calc(cls,def,def)
calc(cls,3,def)
calc(cls,def,4)

ans =
    44
ans =
    14
ans =
    37
Calling .NET Methods

In this section...

“Calling Object Methods” on page 10-80
“Getting Method Information” on page 10-80
“C# Method Access Modifiers” on page 10-80
“VB.NET Method Access Modifiers” on page 10-81
“Reading Method Signatures” on page 10-81

Calling Object Methods

This topic describes information specific to .NET methods. For general information about calling methods on objects in MATLAB, see “Calling Object Methods”.

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
• methodview — Graphical representation of method list

You might find the methodview 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 methodview window for the System.String class:

methodview('System.String')

C# Method Access Modifiers

MATLAB maps C# keywords to MATLAB method access attributes, as shown in the following table.
### Calling .NET Methods

<table>
<thead>
<tr>
<th>C# Method Keyword</th>
<th>MATLAB Attribute</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>RHS, LHS</td>
<td>“Call .NET Methods With ref Keyword” on page 10-73</td>
</tr>
<tr>
<td>out</td>
<td>LHS</td>
<td>“Call .NET Methods With out Keyword” on page 10-71</td>
</tr>
<tr>
<td>params</td>
<td>Array of particular type</td>
<td>“Call .NET Methods With params Keyword” on page 10-75</td>
</tr>
</tbody>
</table>

### VB.NET Method Access Modifiers

MATLAB maps VB.NET keywords to MATLAB method access attributes, as shown in the following table.

<table>
<thead>
<tr>
<th>VB.NET Method Keyword</th>
<th>MATLAB Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByVal</td>
<td>RHS</td>
</tr>
<tr>
<td>Optional</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

### 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.


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 10-77.

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 10-78.

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 10-81. `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.

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:

```matlab
str = System.String('my new string');
methods(str)
```

**Display of System.String Methods**

Methods for class System.String:

- Chars               Normalize           TrimStart
- Clone               PadLeft             addlistener
- CompareTo           PadRight            char
- Contains            Remove              delete
- CopyTo              Replace             eq
- EndsWith            Split               findobj
- Equals              StartsWith          findprop
- GetEnumerator       String              ge
- GetHashCode         Substring           gt
- GetType             ToCharArray         isvalid
- GetTypeCode         ToLower             le
- IndexOf             ToLowerInvariant    lt
- IndexOfAny          ToString            ne
- Insert              ToUpper             notify
- IsNormalized        ToUpperInvariant   op_Equality
- LastIndexOf         Trim
- LastIndexOfAny      TrimEnd

Static methods:

- Compare              Intern               op_Inequality
- CompareOrdinal       IsInterned          op_Equality
- Concat               IsNullOrEmpty
- Copy                 IsNullOrWhiteSpace
- Format               Join

Notice that MATLAB displays the Chars property as a method.

The Chars method has the following signature.
<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char scalar RetVal</td>
<td>Chars</td>
<td>(System.String this, int32 scalar index)</td>
</tr>
</tbody>
</table>

To see the first character, type:

```matlab
Chars(str,0)
```

```matlab
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 http://msdn.microsoft.com/en-us/library/2sk3x8a7(VS.71).aspx on the Microsoft Developer Network website.

<table>
<thead>
<tr>
<th>C++ operator symbol</th>
<th>.NET operator</th>
<th>MATLAB methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (binary)</td>
<td>op_Addition</td>
<td>plus, +</td>
</tr>
<tr>
<td>- (binary)</td>
<td>op_Subtraction</td>
<td>minus, -</td>
</tr>
<tr>
<td>* (binary)</td>
<td>op_Multiply</td>
<td>mtimes, *</td>
</tr>
<tr>
<td>/</td>
<td>op_Division</td>
<td>mrdivide, /</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>op_LogicalAnd</td>
<td>and, &amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>op_Equality</td>
<td>eq, ==</td>
</tr>
<tr>
<td>&gt;</td>
<td>op_GreaterThan</td>
<td>gt, &gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>op_LessThan</td>
<td>lt, &lt;</td>
</tr>
<tr>
<td>!=</td>
<td>op_Inequality</td>
<td>ne, ~=</td>
</tr>
<tr>
<td>&gt;=</td>
<td>op_GreaterThanOrEqual</td>
<td>ge, &gt;=</td>
</tr>
<tr>
<td>&lt;=</td>
<td>op_LessThanOrEqual</td>
<td>le, &lt;=</td>
</tr>
<tr>
<td>- (unary)</td>
<td>op_UnaryNegation</td>
<td>uminus, -a</td>
</tr>
<tr>
<td>+ (unary)</td>
<td>op_UnaryPlus</td>
<td>uplus, +a</td>
</tr>
</tbody>
</table>
Limitations to Support of .NET 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 for Your Class”
• “Methods That Modify Default Behavior”
Use .NET Events in MATLAB

These examples use the `addlistener` function to handle .NET events with MATLAB callbacks.

<table>
<thead>
<tr>
<th>In this section...</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>“Monitor Changes to .TXT File” on page 10-89</td>
<td></td>
</tr>
<tr>
<td>“Monitor Changes to Windows Form ComboBox” on page 10-89</td>
<td></td>
</tr>
</tbody>
</table>

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`:

```matlab
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.

```matlab
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:

```matlab
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.

```matlab
```
To create this example, you must build a Windows Forms Application using a supported version of Microsoft Visual Studio.

- For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.

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.

Create the following MATLAB class, `EnterComboData`, which uses the `attachListener` method to add a listener to the `form` property.

```matlab
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`. Use the `EnterComboData` class to create a form and call its `attachListener` method:
form = EnterComboData;  
form.attachListener;

To trigger an event, select an item from the drop-down menu on the ComboBox.  
MATLAB displays:

Field updated
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). For another example, see “Create Delegates from .NET Object Methods” on page 10-94.

This example consists of the following tasks:

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Declare a Delegate in a C# Assembly” on page 10-92</td>
</tr>
<tr>
<td>“Load the Assembly Containing the Delegate into MATLAB” on page 10-92</td>
</tr>
<tr>
<td>“Select a MATLAB Function” on page 10-92</td>
</tr>
<tr>
<td>“Create an Instance of the Delegate in MATLAB” on page 10-93</td>
</tr>
<tr>
<td>“Invoke the Delegate Instance in MATLAB” on page 10-93</td>
</tr>
</tbody>
</table>

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 10-23.

Load the Assembly Containing the Delegate into MATLAB

If the NetDocDelegate assembly is in your c:\work folder, load the file with the command:

```matlab
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 (string), 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)
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

```csharp
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:

```matlab
NET.addAssembly(dllPath);
NET.addAssembly('c:\work\Recipe.dll');
myRec = Recipe.MyClass;
myFunc = NetDocDelegate.delInteger(@myRec.AddEggs);
myFunc(2)
ans = Add 2 eggs
```
Create Delegate Instances Bound to .NET Methods

For a C# delegate defined as:

```csharp
namespace MyNamespace
{
    public delegate void MyDelegate();
}
```

MATLAB creates the following constructor signature.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyNamespace.MyDelegate obj</td>
<td>MyDelegate</td>
<td>(target, string methodName)</td>
</tr>
</tbody>
</table>

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:

```csharp
namespace MyNamespace
{
    public delegate void MyDelegate();

    public class MyClass
    {
        public void MyMethod()
        {
        }
    }
}
```

To instantiate the delegate in MATLAB, type:

```matlab
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:

```csharp
namespace MyNamespace
{
    public delegate void MyDelegate();

    public class MyClass
    {
        public static void MyStaticMethod()
        {
        }
    }
}
```

To instantiate the delegate in MATLAB, type:

```matlab
delegate = MyNamespace.MyDelegate('MyNamespace.MyClass','MyStaticMethod');
```
Call Delegates With out and ref Type Arguments

The MATLAB rules for mapping out and ref types for delegates are the same as for methods. See “C# Method Access Modifiers” on page 10-80.

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)
```
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 10-20.

For example, create the following MATLAB functions to use with the `NetDocDelegate.delInteger` delegate:

```matlab
function out = action1(n)
out = 'Add flour';
disp(out)
end
```

```matlab
function out = action2(n)
out = 'Add eggs';
disp(out)
end
```

Create delegates `step1` and `step2`:

```matlab
step1 = NetDocDelegate.delInteger(@action1);
step2 = NetDocDelegate.delInteger(@action2);
```

To combine into a new delegate, `mixItems`, type:

```matlab
mixItems = step1.Combine(step2);
```

Or, type:

```matlab
mixItems = step1.Combine(@action2);
```

Invoke `mixItems`:

```matlab
result = mixItems(1);
```

In this case, the function `action2` follows `action1`:

```matlab
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);
.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 10-81 in Using a .NET Object. 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.
Calling .NET Methods Asynchronously

In this section...

“How MATLAB Handles Asynchronous Method Calls in .NET” on page 10-101
“Calling a Method Asynchronously Using a Callback When an Asynchronous Call
Finishes” on page 10-101
“Calling a Method Asynchronously Without a Callback” on page 10-103
“Using EndInvoke With out and ref Type Arguments” on page 10-104
“Using Polling to Detect When Asynchronous Call Finishes” on page 10-104

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” at http://msdn.microsoft.

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. For information about using a callback method, see “Calling a Method Asynchronously Using a Callback When an Asynchronous Call Finishes” on page 10-101.

Note: MATLAB is a single-threaded application. Therefore, handling asynchronous calls in the MATLAB environment might result in deadlocks.

Calling a Method Asynchronously Using a Callback When an Asynchronous Call Finishes

You can execute a callback method when an asynchronous call completes. A callback method executes on a different thread than the thread that processes the results of the asynchronous call.
The following is an overview of the procedure. If you do not use a callback function, follow the procedure in “Calling a Method Asynchronously Without a Callback” on page 10-103.

• Select or create a MATLAB function to execute asynchronously.
• Select or create a C# delegate and associate it with the MATLAB function.
• Create a MATLAB callback function with a System.AsyncCallback Delegate delegate signature. The signature, shown at the MSDN website, is:

  public delegate void AsyncCallback(IAsyncResult ar)

1. Using MATLAB code, initiate the asynchronous call using the BeginInvoke method, specifying the callback delegate and, if required, object parameters.
2. Continue executing commands in MATLAB.
3. When the asynchronous function completes, MATLAB calls the callback function, which executes the EndInvoke method to retrieve the results.

Callback Example

In this example, create the following MATLAB function to execute asynchronously:

```matlab
function X = DivideFunction(A, B)
    if B ~= 0
        X = A / B;
    else
        errid = 'MyID:DivideFunction:DivisionByZero';
        error(errid, 'Division by 0 not allowed.' )
    end
end
```

Create the following MATLAB function, which executes as the callback when the asynchronous method invocation completes. This function displays the result value of the EndInvoke method.

```matlab
function myCallback(asyncRes)
    result = asyncRes.AsyncDelegate.EndInvoke(asyncRes);
    disp(result)
end
```

Use the del2Integer delegate, defined in the NetDocDelegate assembly:

```csharp
public delegate Int32 del2Integer(Int32 arg1, Int32 arg2);
```
Run the example:

```matlab
% Create the delegate
divDel = NetDocDelegate.del2Integer(@DivideFunction);
A = 10;
B = 5;
% Initiate the asynchronous call.
asyncRes = divDel.BeginInvoke(A,B,@myCallback,[]);
```

MATLAB displays the result: 2

## Calling a Method Asynchronously Without a Callback

The following is an overview of the procedure. If you want to use a callback function, follow the procedure in “Calling a Method Asynchronously Using a Callback When an Asynchronous Call Finishes” on page 10-101.

1. Select or create a MATLAB function to execute asynchronously.
2. Select or create a C# delegate and associate it with the MATLAB function.
3. In MATLAB, initiate the asynchronous call using the `BeginInvoke` method.
4. Continue executing commands in MATLAB.
5. Poll for asynchronous call completion using the MATLAB `pause` function.
6. When the asynchronous function completes, call the `EndInvoke` method to retrieve the results.

### Example Without Callback

In this example, create the following MATLAB function, `myFunction`:

```matlab
% MATLAB function to execute asynchronously
function res = myFunction(strValue)
    res = strValue;
end
```

Use the `delString` delegate, defined in the `NetDocDelegate` assembly:

```csharp
public delegate string delString(string message);
```

In MATLAB, create the delegate, `myDelegate`, define the input values, and start the asynchronous call:
myDelegate = NetDocDelegate.delString(@myFunction);
A = 'Hello';
asyncRes = myDelegate.BeginInvoke(A,[],[]);

The BeginInvoke method returns the object, asyncRes, which you use to monitor the progress of the asynchronous call. Poll for results, using the MATLAB pause function to let MATLAB process the events:

```matlab
while asyncRes.IsCompleted ~= true
   pause(0.01)
end
```

Retrieve and display the results of the asynchronous call:

```matlab
result = myDelegate.EndInvoke(asyncRes);
disp(result)
```

Hello

### 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 “Call Delegates With out and ref Type Arguments” on page 10-97. 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's callback on the main thread, call the MATLAB pause (or a similar) function.
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:

```csharp
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, use the technique described in “Calling a Method Asynchronously Without a Callback” on page 10-103. 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 Bit-Wise Operations on System_ENUM” on page 10-107
“Creating .NET Enumeration Bit Flags” on page 10-107
“Removing a Flag from a Variable” on page 10-108
“Replacing a Flag in a Variable” on page 10-108
“Testing for Membership” on page 10-109

How MATLAB Supports Bit-Wise Operations on System_ENUM

Many .NET languages support bit-wise operations on enumerations defined with the System.Flags attribute. The MATLAB language does not have equivalent operations, and, therefore, provides instance methods for performing bit-wise operations on an enumeration object. The bit-wise 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 bit-wise operators, you can create and test for combinations.

To use bit-wise 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:

```matlab
teamMtgs = bitor(...
    NetDocEnum.MyDays.Friday,...
    NetDocEnum.MyDays.Wednesday);
```

Create a variable `allMtgs` of all days with meetings:

```matlab
allMtgs = bitor(teamMtgs,...
    NetDocEnum.MyDays.Monday);
```

To see which days belong to each variable, type:

```matlab
teamMtgs
allMtgs
```

```matlab
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.

```matlab
thisWeekMtgs = bitxor(allMtgs,NetDocEnum.MyDays.Wednesday)
```

```matlab
thisWeekMtgs =
    Monday, Friday
```

Using a bit-wise 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.

```matlab
teamMtgs = bitor(...
Use Bit Flags with .NET Enumerations

(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:

RemindMe(today)

Meeting today.

See Also
bitand | bitnot | bitor | bitxor
Related Examples

- “NetDocEnum Example Assembly” on page 10-115
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
% 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:length(folders)
    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
.NET Enumerations in MATLAB

MATLAB allows you to work with .NET enumerations using features of the MATLAB enumeration class and some features unique to the .NET Framework.

Terms you should know:

• Enumeration — In MATLAB, a class having a finite set of named instances.
• Enumeration member — Named instance of an enumeration class.
• Underlying value — Numeric value associated with an enumeration member.

Enumerations contain the following information:

• Members
• Methods
• Underlying Values

In this topic, the term enumeration refers to a .NET enumeration.

Note: The MATLAB language supports user-defined enumeration classes. If you are using enumerations defined in MATLAB, refer to the topics under Enumerations.
**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.
- Bit-wise methods — Only for enumerations with the `System.Flags` attribute.

For example, type:

```matlab
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 bit-wise methods. To list the methods, type:

```matlab
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

Related Examples
• “Using Relational Operations” on page 10-124
• “Using Switch Statements” on page 10-124
• “Display .NET Enumeration Members as Character Strings” on page 10-120
• “NetDocEnum Example Assembly” on page 10-115

More About
• “Use Bit Flags with .NET Enumerations” on page 10-107
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 10-23.

If the NetDocEnum assembly is in your `c:\work` folder, load the file:

```matlab
dllPath = fullfile('c:\','work','NetDocEnum.dll');
asm = NET.addAssembly(dllPath);
asm.Enums
ans =
 'NetDocEnum.MyDays'
 'NetDocEnum.Range'
```
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:

```matlab
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:

```matlab
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`.

```matlab
whos all*
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>allNames</td>
<td>1x1</td>
<td>112</td>
<td>System.String[]</td>
</tr>
<tr>
<td>allValues</td>
<td>1x1</td>
<td>112</td>
<td>System.DayOfWeek[]</td>
</tr>
</tbody>
</table>

Although the types are different, the information MATLAB displays is the same. For example, type:

```matlab
allNames(1)
```

```matlab
ans =
```
Sunday

Type:

allValues(1)

ans =
Sunday

See Also
enumeration

Related Examples
  • “Iterate Through a .NET Enumeration” on page 10-122

More About
  • “Information About System.Enum Methods” on page 10-122
  • “How MATLAB Handles System.String” on page 10-54
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 =
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1x1</td>
<td>104</td>
<td>NetDocEnum.Range</td>
</tr>
</tbody>
</table>

**Related Examples**

- “NetDocEnum Example Assembly” on page 10-115
Display .NET Enumeration Members as Character Strings

To get the descriptive name of an enumeration, use the `char` method. For example, type:

```matlab
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:

```matlab
gameDay = System.DayOfWeek.Thursday;
myValue = double(gameDay)
```

```
myValue =
 4
```
Iterate Through a .NET Enumeration

In this section...

- “Information About System.Enum Methods” on page 10-122
- “Display Enumeration Member Names” on page 10-123

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:

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System.Type</td>
<td>GetType</td>
<td>(System.Enum this)</td>
</tr>
<tr>
<td>Static</td>
<td>System.String[]</td>
<td>GetNames</td>
<td>(System.Type enumType)</td>
</tr>
<tr>
<td>Static</td>
<td>System.Array</td>
<td>GetValues</td>
<td>(System.Type enumType)</td>
</tr>
</tbody>
</table>

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:

```matlab
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
Use .NET Enumerations to Test for Conditions

**In this section...**

“Using Switch Statements” on page 10-124
“Using Relational Operations” on page 10-124

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:

```matlab
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:

```matlab
today = System.DayOfWeek.Wednesday;
Reminder(today)
```

ans =
Team meeting at 2:00

### Using Relational Operations

Create the following function to display a message:

```matlab
function VolleyballMessage(day)
```
Use .NET Enumerations to Test for Conditions

```matlab
% day = System.DayOfWeek enumeration value
if gt(day, System.DayOfWeek.Thursday)
    disp('See you next week at volleyball.')
else
    disp('See you Thursday!')
end

disp('See you Thursday!')

disp('See you next week at volleyball.')

myDay = System.DayOfWeek.Monday;
VolleyballMessage(myDay)

myDay = System.DayOfWeek.Friday;
VolleyballMessage(myDay)```

For a day before Thursday:

See you Thursday!

For a day after Thursday:

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`:

```csharp
public enum Range : long {Max = 2147483648L, Min = 255L}
```

To display the underlying type:

```csharp
maxValue = NetDocEnum.Range.Max;
System.Enum.GetUnderlyingType(maxValue.GetType).FullName
```

```csharp
ans = System.Int64
```

**Related Examples**

- “NetDocEnum Example Assembly” on page 10-115
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:

```matlab
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 `mscorlib` assembly, which MATLAB loads automatically.

```matlab
    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.

```matlab>AddRange(dc,d1);
```

List `dc` now has three items:

```matlab>dc.Count
```

To display the contents, use the `Item` method and zero-based indexing:

```matlab>for i = 1:dc.Count
        disp(dc.Item(i-1))
end
```
Another way to add values is to use the InsertRange method. Insert the d2 array starting at index 1:

```csharp
InsertRange(dc, 1, d2);
```

The size of the array has grown to seven. To display the values, type:

```csharp
for i = 1:dc.Count
    disp(dc.Item(i-1))
end
```

The first item in the d2 array ('Tyrannosaurus') is at index 1 in list dc:

```csharp
System.String.Compare(d2(1), dc.Item(1))
```

The System.String.Compare answer, 0, indicates that the two values are equal.

**External Websites**

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 the list, starting with index 2. Then call ToArray to create a System.String array dArr, and display the results:

```matlab
temp = GetRange(dc,2,3);
dArr = ToArray(temp);
for i = 1:dArr.Length
    disp(dArr(i))
end
```

```
Spinosaurus
Velociraptor
Triceratops
```

To create a MATLAB array D:

```matlab
D = {char(dArr(1)),char(dArr(2)),char(dArr(3))}
```

```
D =
    'Spinosaurus'    'Velociraptor'    'Triceratops'
```

Now you can use D in MATLAB functions. For example, if you type:

```
D'
```

```
ans =
    'Spinosaurus'
    'Velociraptor'
    'Triceratops'
```

Sort the array alphabetically:

```
sort(D)
```

```
ans =
    'Spinosaurus'    'Triceratops'    'Velociraptor'
```
Create .NET Arrays of Generic Type

This example creates a .NET array of List<Int32> generic type.

```csharp
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 10-132
“Display Generic Methods in a Class” on page 10-133
“Display Generic Methods in a Generic Class” on page 10-134

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
```
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
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
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');
as = 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)'

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 10-137.

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:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.String</td>
<td>Item</td>
<td>(System.Collections.Generic.List&lt;System*String&gt; this, int32 scalar index)</td>
</tr>
</tbody>
</table>

The syntax to use Item to set a value is:

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Item</td>
<td>(System.Collections.Generic.List&lt;System*String&gt; this, int32 scalar index, System.String value)</td>
</tr>
</tbody>
</table>

Related Examples

- “Create .NET Collections” on page 10-128
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 10-20.

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.

In this section...

“Using the NetDocGeneric Example” on page 10-137
“Invoke Generic Class Member Function” on page 10-138
“Invoke Static Generic Functions” on page 10-138
“Invoke Static Generic Functions of a Generic Class” on page 10-138
“Invoke Generic Functions of a Generic Class” on page 10-139

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 10-23.

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 10-132 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`:

```matlab
cls = NetDocGeneric.SampleClass();
```

To convert 5 to an integer parameter type, such as `System.Int32`, call `NET.invokeGenericMethod` with the object:

```matlab
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:

```matlab
ret = NET.invokeGenericMethod(cls,'GenMethodWithMixedArgs',...
    {'System.Double'},5,6,true);
```

To return `arg2`, use the syntax:

```matlab
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:

```matlab
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:

```matlab
genClsDef = NET.GenericClass('NetDocGeneric.SampleGenericClass',...
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.
Using COM Objects from MATLAB

- “MATLAB COM Integration” on page 11-2
- “Registering Controls and Servers” on page 11-5
- “Getting Started with COM” on page 11-7
- “Use Internet Explorer in MATLAB Figure” on page 11-10
- “Add Grid ActiveX Control in a Figure” on page 11-15
- “Read Spreadsheet Data Using Excel as Automation Server” on page 11-23
- “Supported Client/Server Configurations” on page 11-30
MATLAB COM Integration

In this section...

“Concepts and Terminology” on page 11-2
“COM Objects, Clients, and Servers” on page 11-2
“Interfaces” on page 11-3
“The MATLAB COM Client” on page 11-3
“The MATLAB COM Automation Server” on page 11-4

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.

A Microsoft ActiveX® control is a type of in-process COM server that requires a control container. ActiveX controls typically have a user interface. An example is the Microsoft Calendar control. A control container is an application capable of hosting ActiveX controls. A MATLAB figure window or a Simulink model are examples of control containers.

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.

The 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.

In a typical scenario, MATLAB creates ActiveX controls in figure windows, which are manipulated by MATLAB through the controls' properties, methods, and events. This is useful because there exists a wide variety of graphical user interface components implemented as ActiveX controls. For example, the Microsoft Internet Explorer program exposes objects that you can include in a figure to display an HTML file. There also are treeviews, spreadsheets, and calendars available from a variety of sources.

MATLAB COM clients can access applications that support Automation, such as the Excel spreadsheet program. In this case, MATLAB creates an Automation server in which to run the application and returns a handle to the primary interface for the object created.
The 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.

**More About**

- “Create COM Objects” on page 12-3
- “MATLAB COM Automation Server”
Registering Controls and Servers

Before using COM objects, you must register their controls and servers. Most are registered by default. However, if you get a new .ocx, .dll, or other object file for the control or 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
```

For example, to register the MATLAB control mwsamp2.ocx, type:

```
cd matlabroot\toolbox\matlab\winfun\win32
regsvr32 mwsamp2.ocx
```

If you encounter problems with this procedure, consult a Windows manual or contact your local system administrator.

Accessing 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.

Verifying the Registration

Here are several ways to verify that a control or server is registered. These examples use the MATLAB mwsamp control. Refer to your Microsoft product documentation for information about using Microsoft Visual Studio or the Microsoft Registry Editor programs.

- Go to the Visual Studio .NET 2003 Tools menu and execute the ActiveX control test container. Click Edit, insert a new control, and select MwSamp Control. If you are able to insert the control without any problems, the control is successfully registered. This method only works on controls.

- Open the Registry Editor by typing rege edit at the DOS prompt. Search for your control or server object by selecting Find from the Edit menu. It will likely be in the following structure:

```
HKEY_CLASSES_ROOT/progid
```
• Open OLEViewer from the Visual Studio .NET 2003 Tools menu. Look in the following structure for your Control object:

Object Classes : Grouped by Component Category : Control : Your_Control_Object_Name (i.e. Object Classes : Grouped by Component Category : Control : Mwsamp Control)
Creating an Instance of a COM Object

Two MATLAB functions enable you to create COM objects:

• `actxcontrol` — Creates an instance of a control in a MATLAB figure.
• `actxserver` — Creates and manipulates objects from MATLAB that are exposed in an application that supports Automation.

Each 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.

Getting Information About a Particular COM Control

In general, you can determine what you can do with an object using the `methods`, `get`, and `events` functions.

Information about Methods

To list the methods supported by the object `handle`, type:

`methods(handle)`

Information about Properties

To list the properties of the object `handle`, type:

`get(handle)`

To see the value of the property `PropertyName`, type:

`get(handle, 'PropertyName')`
Use the `set` function to change a property value.

**Information about Events**

To list the events supported by the object `handle`, type:

```
events(handle)
```

**Getting an Object’s ProgID**

To get the programmatic identifier (ProgID) of a COM control that is already registered on your computer, use the `actxcontrollist` command. You can also use the ActiveX Control Selector, displayed with the command `actxcontrolselect`. This interface lets you see instances of the controls installed on your computer.

**Registering a Custom Control**

If your MATLAB program uses a custom control (e.g., 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. You might also want to supply versions of a Microsoft ActiveX control to ensure that all users have the same version.

**See Also**

`actxcontrol` | `actxcontrollist` | `actxcontrolselect` | `actxserver` | `events` | `get` | `methods` | `set`

**Related Examples**

- “Creating an ActiveX Control” on page 12-4

**More About**

- “Registering Controls and Servers” on page 11-5
• “COM Methods” on page 12-24
• “COM Events” on page 12-27
• “COM Object Interfaces” on page 12-32
Use Internet Explorer in MATLAB Figure

This example uses the ActiveX control `Shell.Explorer`, which is exposed by the Microsoft Internet Explorer application, to include an HTML viewer in a MATLAB figure. The figure's window button down function is then used to select a graphics object when the user clicks the graph and load the object's property documentation into the HTML viewer.

**Techniques Demonstrated**

- Using Internet Explorer from an ActiveX client program.
- Defining a window button down function that displays HTML property documentation for whatever object the user clicks.
- Defining a `resize` function for the figure that also resizes the ActiveX object container.

**Using the Figure to Access Properties**

This example creates a larger than normal figure window that contains an axes object and an HTML viewer on the lower part of the figure window. By default, the viewer displays the URL `http://www.mathworks.com`. When you issue a plotting command, such as:

```matlab
surf(peaks(20))
```

the graph displays in the axes.

Click anywhere in the graph to see the property documentation for the selected object.
Complete Code Listing

You can open the file that implements this example in MATLAB Editor or you can run this example with the following links:

- Open file in editor
- Run this example
Creating the Figure

This example defines the figure size based on the default figure size and adds space for the ActiveX control. Here is the code to define the figure:

```matlab
dfpos = get(0,'DefaultFigurePosition');
hfig = figure('Position',dfpos([1 2 3 4]).* [.8 .2 1 1.65],...
    'Menu','none','Name','Create a plot and click on an object',...
    'ResizeFcn',@reSize,...
    'WindowButtonDownFcn',@wbdf,...
    'Renderer','Opengl');
```

Note that the figure also defines a resize function and a window button down function by assigning function handles to the `ResizeFcn` and `WindowButtonDownFcn` properties. The callback functions `reSize` and `wbdf` are defined as nested functions in the same file.

Calculating the ActiveX Object Container Size

The `actxcontrol` function creates the ActiveX control inside the specified figure and returns the control's handle. You need to supply the following information:

- Control's programmatic identifier (use `actxcontrollist` to find it)
- Location and size of the control container in the figure (pixels) [left bottom width height]
- Handle of the figure that contains the control:

```matlab
conSize = calcSize;                                  % Calculate the container size
hExp = actxcontrol('Shell.Explorer.2',conSize,hfig);  % Create the control
Navigate(hExp,'http://www.mathworks.com/');           % Specify content of html viewer
```

The nested function, `calcSize` calculates the size of the object container based on the current size of the figure. `calcSize` is also used by the figure `resize` function, which is described in the next section.

```matlab
function conSize = calcSize
fp = get(hfig,'Position');                            % Get current figure size
conSize = [0 0 1 .45].*fp([3 4 3 4]);                 % Calculate container size
end % calcSize
```
Automatic Resize

In MATLAB, you can change the size of a figure and the axes automatically resize to fit the new size. This example implements similar resizing behavior for the ActiveX object container within the figure using the object's `move` method. This method enables you to change both size and location of the ActiveX object container (i.e., it is equivalent to setting the figure `Position` property).

When you resize the figure window, the MATLAB software automatically calls the function assigned to the figure's `ResizeFcn` property. This example implements the nested function `reSize` for the figure `reSize` function.

ResizeFcn at Figure Creation

The `resize` function first determines if the ActiveX object exists because the MATLAB software calls the figure `resize` function when the figure is first created. Since the ActiveX object has not been created at this point, the `resize` function simply returns.

When the Figure Is Resized

When you change the size of the figure, the `resize` function executes and does the following:

- Calls the `calcSize` function to calculate a new size for the control container based on the new figure size.
- Calls the control's `move` method to apply the new size to the control.

Figure ResizeFcn.

```matlab
function reSize(~,~)
if ~exist('hExp','var')
    return
end
conSize = calcSize;
move(hExp,conSize);
end % reSize
```

Selecting Graphics Objects

This example uses the figure `WindowButtonDownFcn` property to define a callback function that handles mouse click events within the figure. When you click the left
Using COM Objects from MATLAB

mouse button while the cursor is over the figure, the MATLAB software executes the WindowButtonDownFcn callback on the mouse down event.

The callback determines which object was clicked by querying the figure CurrentObject property, which contains the handle of the graphics object most recently clicked. Once you have the object's handle, you can determine its type and then load the appropriate HTML page into the Shell.Explorer control.

The nested function wbdf implements the callback. Once it determines the type of the selected object, it uses the control Navigate method to display the documentation for the object type.

**Figure WindowButtonDownFcn.**

```matlab
function wbdf(~,~)
    cobj = get(hfig,'CurrentObject');
    if isempty(cobj)
        disp('Click somewhere else')
        return
    end
    pth = 'http://www.mathworks.com/access/helpdesk/help/techdoc/ref/';
    typ = get(cobj,'Type');
    switch typ
        case ('figure')
            Navigate(hExp,[pth,'figure_props.html']);
        case ('axes')
            Navigate(hExp,[pth,'axes_props.html']);
        case ('line')
            Navigate(hExp,[pth,'line_props.html']);
        case ('image')
            Navigate(hExp,[pth,'image_props.html']);
        case ('patch')
            Navigate(hExp,[pth,'patch_props.html']);
        case ('surface')
            Navigate(hExp,[pth,'surface_props.html']);
        case ('text')
            Navigate(hExp,[pth,'text_props.html']);
        case ('hggroup')
            Navigate(hExp,[pth,'hggroupproperties.html']);
        otherwise
            Navigate(hExp,[pth(1:end-4),'infotool/hgprop/doc_frame.html']);
    end
end % wbdf
```
Add Grid ActiveX Control in a Figure

This example adds a Microsoft ActiveX spreadsheet control to a figure, which also contains an axes object for plotting the data displayed by the control. Clicking a column in the spreadsheet causes the data in that column to be plotted. Clicking down and dragging the mouse across multiple columns plots all columns touched.

Techniques Demonstrated

• Registering a control for use on your system.
• Writing a handler for one of the control's events and using the event to execute MATLAB plotting commands.
• Writing a resize function for the figure that manages the control's size as users resize the figure.

Using the Control

This example assumes that your data samples are organized in columns and that the first cell in each column is a title, which is used by the legend. See “Complete Code Listing” on page 11-16 for an example of how to load data into the control.

Once the data is loaded, click the column to plot the data. The following picture shows a graph of the results of Test2 and Test3 created by selecting column B and dragging and releasing on column C.
Complete Code Listing

You can open the file used to implement this example in MATLAB Editor:

- Open file in editor.
Preparing to Use the Control

The ActiveX control used in this example is typical of those downloadable from the Internet. Once you have downloaded the files you need, register the control on your system using the Windows command `regsvr32`. In a command prompt, enter a command of the following form:

```
regsvr32 sgrid.ocx
```

From the MATLAB command line, type:

```
system 'regsvr32 sgrid.ocx'
```

See the section “Registering Controls and Servers” on page 11-5 for more information.

Finding the Control's ProgID

Once you have installed and registered the control, you can obtain its programmatic identifier using the ActiveX Control Selector dialog. To display this dialog box, use the `actxcontrolselect` command. Locate the control in the list and the selector displays the control and the ProgID.
Creating a Figure to Contain the Control

This example creates a figure that contains an axes object and the grid control. The first step is to determine the size of the figure and then create the figure and axes. This example uses the default figure and axes size (obtained from the respective Position properties) to calculate a new size and location for each object.

```matlab
dfpos = get(0,'DefaultFigurePosition');
dapos = get(0,'DefaultAxesPosition');
hfig = figure('Position',dfpos([1 2 3 4]).*[1 .8 1 1.25],...
    'Name','Select the columns to plot',...
    'Renderer','ZBuffer',...
    'ResizeFcn',{@reSize dfpos(3)});
hax = axes('Position',dapos([1 2 3 4]).*[1 4 1 .65]);
```
The above code moves the figure down from the top of the screen (multiply second element in position vector by 0.8) and increases the height of the figure (multiply fourth element in position vector by 1.25). Axes are created and sized in a similar way.

Creating an Instance of the Control

Use the `actxcontrol` function to create an instance of the control in a figure window. This function creates a container for the control and enables you to specify the size of this container, which usually defines the size of the control. See “Managing Figure Resize” on page 11-21 for a specific example.

Specifying the Size and Location

The control size and location in the figure is calculated by a nested function `calcSize`. This function is used to calculate both the initial size of the control container and the size resulting from resize of the figure. It gets the figure’s current position (i.e., size and location) and scales the four-element vector so that the control container is

- Positioned at the lower-left corner of the figure.
- Equal to the figure in width.
- Has a height that is 0.35 times the figure’s height.

The value returned is of the correct form to be passed to the `actxcontrol` function and the control’s `move` method.

```matlab
function conSize = calcSize
    fp = get(hfig,'Position');
    conSize = fp([3 4 3 4]).*[0 0 1 .35];
end % conSize
```

Creating the Control

Creating the control entails the following steps:

- Calculating the container size
- Instantiating the control in the figure
- Setting the number of rows and columns to match the size of the data array
- Specifying the width of the columns

```matlab
conSize = calcSize;
```
Using COM Objects from MATLAB

hgrid = actxcontrol('SGRID.SgCtrl.1',conSize,hfig);
hgrid.NRows = size(dat,1);
hgrid.NColumns = size(dat,2);
colwth = 4350; hdwth = hgrid.HdrWidth;
SetColWidth(hgrid,0,sz(2)-1,colwth,1)

Using Mouse-Click Event to Plot Data

This example uses the control's Click event to implement interactive plotting. When a user clicks the control, the MATLAB software executes a function that plots the data in the column where the mouse click occurred. Users can also select multiple columns by clicking down and dragging the cursor over more than one column.

Registering the Event

You need to register events with MATLAB so that when the event occurs (a mouse click in this case), the MATLAB software responds by executing the event handler function. Register the event with the registerevent function:

registerevent(hgrid,{'Click',@click_event});

Pass the event name (Click) and a function handle for the event handler function inside a cell array.

Defining the Event Handler

The event handler function click_event uses the control's GetSelection method to determine what columns and rows have been selected by the mouse click. This function plots the data in the selected columns as lines, one line per column.

It is possible to click down on a column and drag the mouse to select multiple columns before releasing the mouse. In this case, each column is plotted because the event is not fired until the mouse button is released (which reflects the way the author chose to implement the control). The legend function uses the column number stored in the variable cols to label the individual plotted lines. You must add one to cols because the control counts the columns starting from zero.

Note that you implement event handlers to accept a variable number of arguments (varargin).

function click_event(varargin)
[ro1,col1,ro2,col2] = hgrid.GetSelection(1,1,1,1,1);
ncols = (col2-col1)+1;
cols = [col1:col2];
  for n = 1:ncols
      hgrid.Col = cols(n);
      for ii = 1:sz(1)
          hgrid.Row = ii;
          plot_data(ii,n) = hgrid.Number;
      end
  end
hgrid.SetSelection(row1,col1,row2,col2);
plot(plot_data)
legend(labels(cols+1))
end % click_event

Managing Figure Resize

The size and location of a MATLAB axes object is defined in units that are normalized to the figure that contains it. Therefore, when you resize the figure, the axes automatically resize proportionally. When a figure contains objects that are not contained in axes, you are responsible for defining a function that manages the resizing process.

The figure ResizeFcn property references a function that executes whenever the figure is resized and also when the figure is first created. This example creates a resize function that manages resizing grid control by doing the following:

- Disables control updates while changes are being made to improve performance (use the hDisplay property).
- Calculates a new size for the control container based on the new figure size (calcSize function).
- Applies the new size to the control container using its move method.
- Scales the column widths of the grid proportional to the change in width of the figure (SetColWidth method).
- Refreshes the display of the control, showing the new size.

function reSize(src,evnt,dfp)
% Return if control does not exist (figure creation)
if ~exist('hgrid','var')
    return
end
% Resize container
hgrid.bDisplay = 0;
conSize = calcSize;
move(hgrid,conSize);
% Resize columns
scl = conSize(3)/dfp;
ncolwth = scl*colwth;
nhdrwth = hdwth*(scl);
hgrid.HdrWidth = nhdrwth;
SetColWidth(hgrid,0,sz(2)-1,ncolwth,2)
hgrid.Refresh;
end % reSize

**Closing the Figure**

This example uses the figure delete function (*DeleteFcn* property) to delete the ActiveX object before closing the figure. The MATLAB software calls the figure delete function before deleting the figure, which enables the function to perform any clean up needed before closing the figure. The figure delete function calls the control's `delete` method.

```matlab
function figDelete(src, evnt)
    delete(hgrid);
end
```
Read Spreadsheet Data Using Excel as Automation Server

For alternatives to importing Excel spreadsheet data into MATLAB, see the functions and examples in “Spreadsheets”.

This example creates a user interface to access the data in a Microsoft Excel file. To enable the communication between MATLAB and the spreadsheet program, this example creates a Microsoft ActiveX 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.

Techniques Demonstrated

• Use of an Automation server to access another application from MATLAB.
• Ways to manipulate Excel data into types used in the UI and plotting.
• Implementing a UI that enables plotting of selected columns of the Excel spreadsheet.
• Inserting a MATLAB figure into an Excel file.

Using the UI

To use the UI, 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 which are plotted versus the first column in the Excel file, which is the time data.

You can view the Excel data file by clicking the Show Excel Data File button, and you can save an image of the graph in a different Excel file by clicking Save Graph button. Note that the Save Graph option creates a temporary PNG file in the current folder, if you have write-access permission.

The following picture shows the UI with an input/response pair selected in the list box and plotted in the axes.
Complete Code Listing

You can open the file used to implement this example in MATLAB Editor or run this example:

- Open file in editor.
- Run this example.

Excel Spreadsheet Format

This example assumes a particular organization of the Excel spreadsheet, as shown in the following picture.
The format of the Excel file is as follows:

- The first element in each column is a text string that identifies the data contained in the column. These strings 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.

**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`.

```matlab
exl = actxserver('excel.application');
```

The `ActiveX` object that is returned provides access to a number of interfaces supported by the Excel program. Use the workbook interface to open the Excel file containing the data.

```matlab
exlWkbk = exl.Workbooks;
```
Using COM Objects from MATLAB

```matlab
exlFile = exlWkbk.Open([docroot '/techdoc/matlab_external/examples/input_resp_data.xls']);
```

Use the workbook's sheet 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 in column A, first cell to column G, last cell:

```matlab
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. This object returns the data in a MATLAB cell array, which can contain both numeric and character data:

```matlab
exlData = rngObj.Value;
```

**Manipulating the Data in the MATLAB Workspace**

Now that the data is in a cell array, you can use MATLAB functions to extract and reshape parts of the data into the forms needed to use in the UI and pass to the plot function.

The following code performs two operations:

- Extracts numeric data from the cell array (indexing with curly braces), concatenates the individual doubles returned by the indexing operation (square brackets), and reshapest it into an array that arranges the data in columns.
- Extracts the string in the first cell in each column of an Excel sheet and stores them in a cell array, which is used to generate the items in the list box.

```matlab
for ii = 1:size(exlData,2)
    matData(:,ii) = reshape([exlData{2:end,ii}],size(exlData(2:end,ii)));
    lBoxList{ii} = [exlData{1,ii}];
end
```

**The Plotter UI**

This example uses a UI that enables you to select from a list of input and response data from a list box. All data is plotted as a function of time (which is, therefore, not a choice in the list box) and you can continue to add more data to the graph. Each data plot added to the graph causes the legend to expand.

Additional implementation details include:
• A legend that updates as you add data to a graph
• A clear button that enables you to clear all graphs from the axes
• A save button that saves the graph as a PNG file and adds it to another Excel file
• A toggle button that shows or hides the Excel file being accessed
• The figure delete function (DeleteFcn property), which MATLAB calls when the figure is closed, is used to terminate the Automation server process.

Selecting and Plotting 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. The legend is updated to display any new data while maintaining the legend for the existing data.

```matlab
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 lBoxList(iSelected+1)]);
end % plotButtonCallback
```

Clearing 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 the UI has been closed, so this example uses the figure's delete function to terminate the Excel process with the Quit method. Also, terminate the second Excel process used for saving the graph. See “Inserting MATLAB Graphs Into Excel Spreadsheets” on page 11-28 for information on this second process.

%% Terminate Excel processes
function deleteFig(src,evt)
    exlWkbk.Close
    exlWkbk2.Close
    exl.Quit
    exl2.Quit
end % deleteFig

Inserting MATLAB Graphs Into Excel Spreadsheets

You can save the graph created with this UI 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 UI initialization phase:

```matlab
exl2 = actxserver('excel.application');
exlWkbk2 = exl2.Workbooks;
wb = invoke(exlWkbk2,'Add');
graphSheet = invoke(wb.Sheets,'Add');
Shapes = graphSheet.Shapes;
```

The following code implements the **Save Graph** button callback:

```matlab
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
```

**See Also**

`xlsread`

**More About**

• “Ways to Import Spreadsheets”
Supported Client/Server Configurations

In this section...

“Introduction” on page 11-30
“MATLAB Client and In-Process Server” on page 11-30
“MATLAB Client and Out-of-Process Server” on page 11-31
“COM Implementations Supported by MATLAB Software” on page 11-32
“Client Application and MATLAB Automation Server” on page 11-32
“Client Application and MATLAB Engine Server” on page 11-33

Introduction

You can configure MATLAB software 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 13-5.

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 12-32.

Microsoft ActiveX Controls

An ActiveX control is an object with a user interface. When MATLAB constructs an ActiveX control, it places the control’s UI in a MATLAB figure window. Click the
various options available in the user interface (e.g., making a particular menu selection) to trigger *events* that get communicated from the control in the server to the client MATLAB application. The client decides what to do about each event and responds accordingly.

MATLAB comes with a sample ActiveX control called *mwsamp*. This control draws a circle on the screen and displays some text. You can use this control to try out MATLAB COM features. For more information, see “MATLAB Sample ActiveX Control *mwsamp*” on page 12-54.

**DLL Servers**

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.

MATLAB responds to events generated by a DLL server in the same way as events from an ActiveX control.

**For More Information**

To learn more about working with MATLAB as a client, see “Create COM Objects” on page 12-3.

**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 13-5. 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 12-32.
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.

MATLAB responds to events generated by an out-of-process server in the same way as events from an ActiveX control.

**For More Information**

To learn more about working with MATLAB as a client, see “Create COM Objects” on page 12-3.

**COM Implementations Supported by MATLAB Software**

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 13-4 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 13-4.

**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.

![Diagram of COM Client and COM Server](image)

Use the MATLAB Engine API functions to start and end the server process, and to send commands to be processed by MATLAB.

**For More Information**

To learn more about the MATLAB engine and the functions provided in its C/C++ and Fortran API Reference libraries, see “MATLAB Engine API for C, C++, and Fortran”.
MATLAB COM Client Support

- “Create COM Objects” on page 12-3
- “Handle COM Data in MATLAB” on page 12-12
- “COM Object Properties” on page 12-21
- “COM Methods” on page 12-24
- “COM Events” on page 12-27
- “COM Event Handlers” on page 12-29
- “COM Object Interfaces” on page 12-32
- “Save and Delete COM Objects” on page 12-35
- “MATLAB Application as DCOM Client” on page 12-37
- “Explore COM Objects” on page 12-38
- “Change Row Height in Range of Spreadsheet Cells” on page 12-42
- “Write Data to Excel Spreadsheet Using ActiveX” on page 12-44
- “Change Cursor in Spreadsheet” on page 12-46
- “Insert Spreadsheet After First Sheet” on page 12-47
- “Redraw Circle in mwsamp Control” on page 12-48
- “Connect to Existing Excel Application” on page 12-50
- “Display Message for Workbook OnClose Event” on page 12-51
- “Run Macro in Excel Server Application” on page 12-52
- “Combine Event Handlers as MATLAB Local Functions” on page 12-53
- “MATLAB Sample ActiveX Control mwsamp” on page 12-54
- “Display Event Messages from mwsamp Control” on page 12-55
- “Add Position Property to mwsamp Control” on page 12-58
- “Save mwsamp2 COM Control” on page 12-59
- “Deploy ActiveX Controls Requiring Run-Time Licenses” on page 12-60
- “Microsoft Forms 2.0 Controls” on page 12-62
• “COM Collections” on page 12-64
• “MATLAB COM Support Limitations” on page 12-65
• “Interpreting Argument Callouts in COM Error Messages” on page 12-66
Create COM Objects

In this section...

“Creating the Server Process — An Overview” on page 12-3
“Creating an ActiveX Control” on page 12-4
“Creating a COM Server” on page 12-10

Creating the Server Process — An Overview

MATLAB software provides two functions to create a COM object:

- `actxcontrol` — Creates a Microsoft ActiveX control in a MATLAB figure window.
- `actxserver` — Creates an in-process server for a dynamic link library (DLL) component or an out-of-process server for an executable (EXE) component.

The following diagram shows the basic steps in creating the server process. For more information on how the MATLAB software establishes interfaces to the resultant COM object, see “COM Object Interfaces” on page 12-32.
Creating an ActiveX Control

You can create an instance of an ActiveX control from the MATLAB client using either a user interface (actxcontrolselect) or the actxcontrol function from the command line. Either of these methods creates an instance of the control in the MATLAB client process and returns a handle to the primary interface to the COM object. Through this interface, you can access the public properties or methods of the object. You can also establish more interfaces to the object, including interfaces that use IDispatch, and any custom interfaces that might exist.
This section describes how to create an instance of the control and how to position it in the MATLAB figure window.

- “Listing Installed Controls” on page 12-5
- “Finding a Particular Control” on page 12-5
- “Creating Control Objects Using a UI” on page 12-6
- “Creating Control Objects from the Command Line” on page 12-9
- “Repositioning the Control in a Figure Window” on page 12-9
- “Limitations to ActiveX Support” on page 12-9

**Listing Installed Controls**

The `actxcontrollist` function shows you what COM controls are currently installed on your system. Type:

```matlab
list = actxcontrollist
```

MATLAB displays a cell array listing each control, including its name, programmatic identifier (ProgID), and file name.

This example shows information for several controls (your results might be different):

```matlab
list = actxcontrollist;
s = sprintf(' Name = %s
 ProgID = %s
 File = %s', list{114:115,:})
```

MATLAB displays:

```
s =

  Name = OleInstall Class
  ProgID = Outlook Express Mime Editor
  File = OlePrn.OleInstall.1
  Name = OutlookExpress.MimeEdit.1
  ProgID = C:\WINNT\System32\oleprn.dll
  File = C:\WINNT\System32\inetcomm.dll
```

**Finding a Particular Control**

If you know the name of a control, you can find it in the list and display its ProgID and the path of the folder containing it. For example, some of the examples in this documentation use the `Mwsamp2` control. You can find it with the following code:
list = actxcontrollist;
for ii = 1:length(list)
    if ~isempty(strfind([list{ii,:}],'Mwsamp2'))
        s = sprintf(' Name = %s
                   ProgID = %s
                   File = %s', ...
                   list{ii,:})
    end
end

s =

    Name = Mwsamp2 Control
    ProgID = MWSAMP.MwsampCtrl.2
    File = D:\Apps\MATLAB\R2006a\toolbox\matlab\winfun\win32\mwsamp2.ocx

The location of this file depends on your MATLAB installation.

**Creating Control Objects Using a UI**

Using the `actxcontrolselect` function is the simplest way to create an instance of a control object. This function displays all controls installed on your system. When you select an item from the list and click the **Create** button, MATLAB creates the control and returns a handle to it. Type:

h = actxcontrolselect

MATLAB displays the **Select an ActiveX Control** dialog box.
The interface has an **ActiveX Control List** selection pane on the left and a **Preview** pane on the right. To see a preview of the control, click one of the control names in the selection pane. (A blank preview pane means that the control does not have a preview.) If MATLAB cannot create the instance, an error message appears in the preview pane.

**Setting Properties with actxcontrolselect**

To change property values when creating the control, click the **Properties** button in the **Preview** pane. You can select which figure window to put the control in (**Parent** field), where to position it in the window (**X** and **Y** fields), and what size to make the control (**Width** and **Height**).

You can register events you want the control to respond to in this window. Register an event and the callback routine to handle that event by entering the name of the routine to the right of the event under **Event Handler**.
You can select callback routines by clicking a name in the Event column, and then clicking the Browse button. To assign a callback routine to more than one event, first press the Ctrl key and click individual event names, or drag the mouse over consecutive event names, and then click Browse to select the callback routine.

MATLAB only responds to registered events, so if you do not specify a callback, the event is ignored.

For example, in the ActiveX Control List pane, select Calendar Control 10.0 (the version on your system might be different) and click Properties. MATLAB displays the Choose ActiveX Control Creation Parameter dialog box. To change the default size for the control, enter a Width of 500 and a Height of 350. Click OK in this window, and click Create in the next window to create an instance of the Calendar control.

You can also set control parameters using the actxcontrol function. One parameter you can set with actxcontrol, but not with actxcontrolselect, is the name of an initialization file. When you specify this file name, MATLAB sets the initial state of the control to that of a previously saved control.

Information Returned by actxcontrolselect

The actxcontrolselect function creates an object that is an instance of the MATLAB COM class. The function returns up to two arguments: a handle for the object, h, and a 1-by-3 cell array, info, containing information about the control. To get this information, type:

\[
[h, \text{info}] = \text{actxcontrolselect}
\]

The cell array info shows the name, ProgID, and file name for the control.

If you select the Calendar Control, and then click Create, MATLAB displays information like:

\[
h = \\
    \text{COM.mscal.calendar.7} \\
\text{info =} \\
    [1x20 \text{ char}] 'MSCAL.Calendar.7' [1x41 \text{ char}]
\]

To expand the info cell array, type:

\[
\text{info{:}}
\]

MATLAB displays:
ans = 
    Calendar Control 9.0
ans = 
    MSCAL.Calendar.7
ans = 
    D:\Applications\MSOffice\Office\MSCAL.OCX

Creating Control Objects from the Command Line

If you already know which control you want and you know its ProgID, use the `actxcontrol` function to create an instance of the control.

The ProgID is the only required input to this function. However, as with `actxcontrolselect`, you can supply more inputs that enable you to select which figure window to put the control in, where to position it in the window, and what size to make it. You can also register any events you want the control to respond to, or set the initial state of the control by reading that state from a file. See the `actxcontrol` reference page for a full explanation of its input arguments.

The `actxcontrol` function 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 12-32.

Repositioning the Control in a Figure Window

After creating a control, you can change its shape and position in the window with the `move` function.

Observe what happens to the object created in the last section when you specify new origin coordinates (70, 120) and new width and height dimensions of 400 and 350:

`move(h,[70 120 400 350]);`

Limitations to ActiveX Support

A MATLAB COM ActiveX control container does not in-place activate controls until they are visible.
Creating a COM Server

Instantiating a 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 12-32.

Unlike Microsoft ActiveX controls, any user interface displayed by the server appears in a separate window.


Instantiating an 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`, when used to create an executable, is `actxserver(ProgID, sysname)`, where `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 12-32.

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 \( h \).

\[
  h = \text{actxserver('Excel.Application')}
\]

MATLAB displays:

\[
  h = \\
  \quad \text{COM.excel.application}
\]

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 \( h \).

\[
  h = \text{actxGetRunningServer('Excel.Application')}
\]

MATLAB displays:

\[
  h = \\
  \quad \text{COM.excel.application}
\]
Handle COM Data in MATLAB

In this section...

“Passing Data to COM Objects” on page 12-12
“Handling Data from COM Objects” on page 12-14
“Unsupported Types” on page 12-15
“Passing MATLAB Data to ActiveX Objects” on page 12-15
“Passing MATLAB SAFEARRAY to COM Object” on page 12-15
“Reading SAFEARRAY from COM Objects in MATLAB Applications” on page 12-17
“Displaying MATLAB Syntax for COM Objects” on page 12-18

Passing 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 “Handling Data from COM Objects” on page 12-14.

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Closest COM Type</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>VT_DISPATCH</td>
<td>VT_DISPATCH</td>
</tr>
<tr>
<td></td>
<td>VT_UNKNOWN</td>
<td>VT_UNKNOWN</td>
</tr>
<tr>
<td>string</td>
<td>VT_BSTR</td>
<td>VT_LPWSTR</td>
</tr>
<tr>
<td></td>
<td>VT_LPSTR</td>
<td>VT_BSTR</td>
</tr>
<tr>
<td></td>
<td>VT_BSTR</td>
<td>VT_FILETIME</td>
</tr>
<tr>
<td></td>
<td>VT_ERROR</td>
<td>VT_ERROR</td>
</tr>
<tr>
<td></td>
<td>VT_DECIMAL</td>
<td>VT_CLSID</td>
</tr>
<tr>
<td></td>
<td>VT_DATE</td>
<td>VT_DATE</td>
</tr>
<tr>
<td>int16</td>
<td>VT_I2</td>
<td>VT_I2</td>
</tr>
</tbody>
</table>
### MATLAB Argument | Closest COM Type | Allowed Types
--- | --- | ---
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.

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Closest COM Type</th>
<th>Allowed Types</th>
</tr>
</thead>
</table>
| variant | VT_VARIANT | VT_VARIANT
 |  | VT_USERDEFINED
 |  | VT_ARRAY
| variant(pointer) | VT_VARIANT | VT_VARIANT | VT_BYREF

**SAFEARRAY Data**

When a COM method identifies a SAFEARRAY or SAFEARRAY(pointer), the MATLAB equivalent is a matrix.

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Closest COM Type</th>
<th>Allowed Types</th>
</tr>
</thead>
</table>
| SAFEARRAY | VT_SAFEARRAY | VT_SAFEARRAY

12-13
Handling 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.

<table>
<thead>
<tr>
<th>COM Variant Type</th>
<th>Description</th>
<th>MATLAB Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_DISPATCH</td>
<td>IUnknown *</td>
<td>handle</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>IDispatch *</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATLAB does not support the IUnknown and IDispatch interfaces with 64-bit MATLAB.</td>
<td></td>
</tr>
<tr>
<td>VT_LPWSTR</td>
<td>wide null terminated string</td>
<td>string</td>
</tr>
<tr>
<td>VT_LPSTR</td>
<td>null terminated string</td>
<td></td>
</tr>
<tr>
<td>VT_BSTR</td>
<td>OLE Automation string</td>
<td></td>
</tr>
<tr>
<td>VT_FILETIME</td>
<td>FILETIME</td>
<td></td>
</tr>
<tr>
<td>VT_ERROR</td>
<td>SCODE</td>
<td></td>
</tr>
<tr>
<td>VT_DECIMAL</td>
<td>16-byte fixed point Class ID date</td>
<td></td>
</tr>
<tr>
<td>VT_CLSID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VT_DATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VT_INT</td>
<td>signed machine int</td>
<td>double</td>
</tr>
<tr>
<td>VT_UINT</td>
<td>unsigned machine int</td>
<td></td>
</tr>
<tr>
<td>VT_I2</td>
<td>2 byte signed int</td>
<td></td>
</tr>
<tr>
<td>VT_UI2</td>
<td>unsigned short</td>
<td></td>
</tr>
<tr>
<td>VT_I4</td>
<td>4 byte signed int</td>
<td></td>
</tr>
<tr>
<td>VT_UI4</td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>VT_R4</td>
<td>4 byte real</td>
<td></td>
</tr>
<tr>
<td>VT_R8</td>
<td>8 byte real</td>
<td></td>
</tr>
<tr>
<td>VT_CY</td>
<td>currency</td>
<td></td>
</tr>
</tbody>
</table>
Handle COM Data in MATLAB

<table>
<thead>
<tr>
<th>COM Variant Type</th>
<th>Description</th>
<th>MATLAB Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_I8</td>
<td>signed int64</td>
<td>int64</td>
</tr>
<tr>
<td>VT_UI8</td>
<td>unsigned int64</td>
<td>uint64</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td>logical</td>
<td>logical</td>
</tr>
<tr>
<td>VT_I1</td>
<td>signed char</td>
<td>char</td>
</tr>
<tr>
<td>VT_UI1</td>
<td>unsigned char</td>
<td>char</td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>VARIANT *</td>
<td>variant</td>
</tr>
<tr>
<td>VT_USERDEFINIED</td>
<td>user-defined type</td>
<td></td>
</tr>
<tr>
<td>VT_ARRAY</td>
<td>SAFEARRAY*</td>
<td></td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>VT_BYREF</td>
<td>VARIANT *</td>
</tr>
<tr>
<td></td>
<td>void* for local use</td>
<td></td>
</tr>
<tr>
<td>VTSAFEARRAY</td>
<td>use VT_ARRAY in VARIANT</td>
<td>SAFEARRAY</td>
</tr>
<tr>
<td>VTSAFEARRAY</td>
<td>VT_BYREF</td>
<td>SAFEARRAY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsupported Types

MATLAB does not support the following COM interface types and displays the warning ActiveX - unsupported VARIANT type encountered.

- Structure
- Sparse array
- Multidimensional SAFEARRAYs (greater than two dimensions)
- Write-only properties

Passing MATLAB Data to ActiveX Objects

The tables also show the mapping of MATLAB types to COM types that you must use to pass data from MATLAB to an Microsoft ActiveX object. For all other types, MATLAB displays the warning ActiveX - invalid argument type or value.

Passing 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 Software” on page 12-16
• “Examples” on page 12-16
• “How to Pass a Single-Dimension SAFEARRAY” on page 12-17
• “Passing SAFEARRAY by Reference” on page 12-17

Default Behavior in MATLAB Software

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:

```matlab
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:

```matlab
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:

\[
D = \begin{bmatrix}
2 & 3 & 4 & 5 \\
5 & 6 & 7 & 8
\end{bmatrix}
\]

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

How to Pass a Single-Dimension SAFEARRAY

For information about passing arguments as one-dimensional arrays to a COM object, see http://www.mathworks.com/matlabcentral/answers/92424-how-can-i-pass-arguments-to-an-activex-server-from-matlab-7-0-r14-as-one-dimensional-arrays.

Passing SAFEARRAY by Reference


Reading 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. For example, using methods from the MATLAB sample control `mwsamp`, type:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.1')
a = GetI4Vector(h)
```

MATLAB displays:

```matlab
a =
   1    2    3
```

MATLAB reads a two-dimensional SAFEARRAY with \( n \) elements as a 2-by-\( n \) matrix. For example:

```matlab
a = GetR8Array(h)
```

MATLAB displays:

```matlab
a =
   1    2    3
   4    5    6
```

MATLAB reads a three-dimensional SAFEARRAY with two elements as a 2-by-2-by-2 cell array. For example:

```matlab
a = GetBSTRArray(h)
```

MATLAB displays:

```matlab
a(:,:,1) =
   '1 1 1'      '1 2 1'
   '2 1 1'      '2 2 1'

a(:,:,2) =
   '1 1 2'      '1 2 2'
   '2 1 2'      '2 2 2'
```

### Displaying 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.

```matlab
h = actxserver('MyApp');
invoke(h)
```

MATLAB displays:

```matlab
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.

```matlab
var = h.TestMeth1;
```

or:

```matlab
var = TestMeth1(h);
```

While the following syntax is correct, its use is discouraged:

```matlab
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,
Using the `invoke` function, MATLAB displays the list of methods:

```matlab
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`, as well as returning a variable `dret` of type `double`. Some pseudo-code examples of calling `TestMeth2` are:

```matlab
var = h.TestMeth2(5);
```

or:

```matlab
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.

```matlab
[dargout, 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**

**In this section...**

- “MATLAB Functions for Object Properties” on page 12-21
- “Work with Multiple Objects” on page 12-21
- “Enumerated Values for Properties” on page 12-22
- “Property Inspector” on page 12-22
- “Custom Properties” on page 12-23
- “Properties That Take Arguments” on page 12-23

**MATLAB Functions for Object Properties**

You can get the value of a property and, in some cases, 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>List one or more properties and their values.</td>
</tr>
<tr>
<td>set</td>
<td>Set the value of one or more properties.</td>
</tr>
<tr>
<td>isprop</td>
<td>Determine if an item is a property of a COM object.</td>
</tr>
<tr>
<td>addproperty</td>
<td>Add a custom property to a COM object.</td>
</tr>
<tr>
<td>deleteproperty</td>
<td>Remove a custom property from a COM object.</td>
</tr>
<tr>
<td>inspect</td>
<td>Open the Property Inspector to display and modify property values.</td>
</tr>
<tr>
<td>propedit</td>
<td>Display the built-in property page of the control, if any.</td>
</tr>
</tbody>
</table>

**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 given a string 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.

```matlab
h = actxserver('Excel.Application');
h.DefaultFilePath = 'C:\ExcelWork';
```

Display the properties of the object.

```matlab
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.

```matlab
h.DefaultFilePath
```

```matlab
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.
Using the Property Inspector on 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.

Related Examples

• “Change Cursor in Spreadsheet” on page 12-46
• “Add Position Property to mwsamp Control” on page 12-58
• “Change Row Height in Range of Spreadsheet Cells” on page 12-42

More About

• “Exploring Properties” on page 12-38
COM Methods

In this section...

“Getting Method Information” on page 12-24
“Calling Object Methods” on page 12-24
“Specifying Enumerated Parameters” on page 12-25
“Skipping Optional Input Arguments” on page 12-25
“Returning Multiple Output Arguments” on page 12-26

Getting 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>methodsview</td>
<td>Graphical display of function names and signatures</td>
</tr>
<tr>
<td>methods with -full qualifier</td>
<td>Cell array of function names and signatures, sorted alphabetically</td>
</tr>
<tr>
<td>methods</td>
<td>Cell array of function names only, sorted alphabetically, with uppercase names listed first</td>
</tr>
<tr>
<td>invoke</td>
<td>Cell array of function names and signatures</td>
</tr>
</tbody>
</table>

Calling Object Methods

MATLAB supports the following syntaxes to call methods on an object.

- By method name:
  
  \[
  \text{outputvalue} = \text{methodname}(\text{object}, \text{'arg1'}, \text{'arg2'}, \ldots);
  \]
• By dot notation:
  
  \[
  \text{outputvalue} = \text{object.methodname('arg1','arg2',...)};
  \]

• Using explicit syntax:

  \[
  \text{outputvalue} = \text{invoke(object,'methodname','arg1','arg2',...)};
  \]

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 control or 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.

### Specifying 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`.

### Skipping 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:

\[
\text{methodname(handle,arg1,[],arg3)};
\]
Returning Multiple Output Arguments

If a server function supports multiple outputs, you can return any or all of 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.

Related Examples

- “Change Row Height in Range of Spreadsheet Cells” on page 12-42
- “Redraw Circle in mwsamp Control” on page 12-48
- “Insert Spreadsheet After First Sheet” on page 12-47

More About

- “Handle COM Data in MATLAB” on page 12-12
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 Microsoft ActiveX control or 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 control or 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 control or 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.

For ActiveX controls, you can register events at the time you create an instance of the control using the `actxcontrol` function.

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
actxcontrol | eventlisteners | events | registerevent | unregisterallevents | unregisterevent

Related Examples
• “Add Grid ActiveX Control in a Figure” on page 11-15
• “Read Spreadsheet Data Using Excel as Automation Server” on page 11-23
• “Combine Event Handlers as MATLAB Local Functions” on page 12-53

More About
• “COM Event Handlers” on page 12-29
COM Event Handlers

In this section...

“Overview of Event Handling” on page 12-29
“Arguments Passed to Event Handlers” on page 12-29
“Event Structure” on page 12-30

Overview of Event Handling

For controls, register handler functions either at the time you create an instance of the control (using `actxcontrol`), or any time afterward (using `registerevent`).

For servers, use `registerevent` to register 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

<table>
<thead>
<tr>
<th>Arg. No.</th>
<th>Contents</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Object name</td>
<td>MATLAB COM class</td>
</tr>
<tr>
<td>2</td>
<td>Event ID</td>
<td>double</td>
</tr>
<tr>
<td>3</td>
<td>Start of Event Argument List</td>
<td>As passed by the control</td>
</tr>
<tr>
<td>end-2</td>
<td>End of Event Argument List</td>
<td>As passed by the control</td>
</tr>
<tr>
<td>(Argument N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end-1</td>
<td>Event Structure</td>
<td>structure</td>
</tr>
<tr>
<td>end</td>
<td>Event Name</td>
<td>char array</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Event Name</td>
<td>char array</td>
</tr>
<tr>
<td>Source</td>
<td>Control Name</td>
<td>MATLAB COM class</td>
</tr>
<tr>
<td>EventID</td>
<td>Event Identifier</td>
<td>double</td>
</tr>
<tr>
<td>Event Arg Name 1</td>
<td>Event Arg Value 1</td>
<td>As passed by the control</td>
</tr>
<tr>
<td>Event Arg Name 2</td>
<td>Event Arg Value 2</td>
<td>As passed by the control</td>
</tr>
<tr>
<td>etc.</td>
<td>Event Arg N</td>
<td>As passed by the control</td>
</tr>
</tbody>
</table>

For example, when the **MouseDown** event of the **mwsamp2** control is triggered, MATLAB passes this Event Structure to the registered event handler:

```
Type: 'MouseDown'
Source: [1x1 COM.mwsamp.mwsampctrl.2]
EventID: -605
Button: 1
Shift: 0
    x: 27
    y: 24
```

**See Also**

actxcontrol | events | registerevent
Related Examples

• “Display Event Messages from mwsamp Control” on page 12-55
• “Combine Event Handlers as MATLAB Local Functions” on page 12-53

More About

• “COM Events” on page 12-27
COM Object Interfaces

In this section...

“IUnknown and IDispatch Interfaces” on page 12-32
“Custom Interfaces” on page 12-33

IUnknown and IDispatch Interfaces

When you invoke the `actxserver` or `actxcontrol` functions, the MATLAB software 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:

```matlab
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:

```matlab
w = h.Workbooks
```

MATLAB displays:

```matlab
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 Using ActiveX” on page 12-44
- “Read Spreadsheet Data Using Excel as Automation Server” on page 11-23
- `actxserver`

**Custom Interfaces**

The MATLAB COM Interface supports custom interfaces for the following client/server configurations:

- “MATLAB Client and In-Process Server” on page 11-30
- “MATLAB Client and Out-of-Process Server” on page 11-31

Limitations to custom interface support are:

- Custom interfaces are not supported on a 64-bit version of MATLAB.
- You cannot invoke functions with optional parameters.

Once you have created a server, you can query the server component to see if any custom interfaces are implemented using the `interfaces` function.

For example, if you have a component with the ProgID `mytestenv.calculator`, you can see its custom interfaces using the commands:

```matlab
h = actxserver('mytestenv.calculator');
customlist = h.interfaces
```
MATLAB displays the interfaces, which might be:

customlist =
    ICalc1
    ICalc2
    ICalc3

To get the handle to a particular interface, use the invoke function:

c1 = invoke(h,'ICalc1')
c1 =
    Interface.Calc_1.0_Type_Library.ICalc_Interface

Use this handle c1 to access the properties and methods of the object through this custom interface ICalc1.

For example, to list the properties, use:

get(c1)
    background: 'Blue'
    height: 10
    width: 0

To list the methods, use:

invoke(c1)
    Add = double Add(handle, double, double)
    Divide = double Divide(handle, double, double)
    Multiply = double Multiply(handle, double, double)
    Subtract = double Subtract(handle, double, double)

To add and multiply numbers using the Add and Multiply methods of the object, use:

sum = Add(c1,4,7)
sum =
    11

prod = c1.Multiply(4, 7)
prod =
    28
Save and Delete COM Objects

In this section...

“Functions for Saving and Restoring COM Objects” on page 12-35
“Releasing COM Interfaces and Objects” on page 12-35

Functions for Saving and Restoring COM Objects

Use these MATLAB functions to save and restore the state of a COM control object.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Load and initialize a COM control object from a file</td>
</tr>
<tr>
<td>save</td>
<td>Write and serialize a COM control object to a file</td>
</tr>
</tbody>
</table>

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.

Releasing COM Interfaces and Objects

Use these MATLAB functions to release or delete a COM object or interface.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>Delete a COM object or interface</td>
</tr>
<tr>
<td>release</td>
<td>Release a COM interface</td>
</tr>
</tbody>
</table>

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 control or 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 control or 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 control or 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.

**Related Examples**
- “Save mwsamp2 COM Control” on page 12-59
MATLAB Application as DCOM Client

Distributed Component Object Model (DCOM) is a protocol that allows clients to use remote COM objects over a network. Additionally, 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.

In this section...

“Exploring Properties” on page 12-38
“Exploring Methods” on page 12-39
“Exploring Events” on page 12-39
“Exploring Interfaces” on page 12-40
“Identifying Objects and Interfaces” on page 12-40

Exploring Properties

A property is information that is associated with a COM object. To see a list of properties of an object, use the COM 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:

```matlab
myApp = actxserver('Excel.Application');
get(myApp)
```

MATLAB displays the properties for your Excel version. To display a single property, type:

```matlab
myApp.OrganizationName
```

```
ans =
MathWorks, Inc.
```

MATLAB displays the value for your application.

To open the Property Inspector, choose one of the following.

- Call the `inspect` function from the MATLAB command line:

  ```matlab
  inspect(myApp)
  ```
• Double-click the `myApp` object in the MATLAB Workspace browser.

MATLAB opens the Inspector window. Scroll down until you see the `OrganizationName` property, the same value returned by the `get` function.

**Exploring 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:

```matlab
myApp = actxserver('Excel.Application');
methodsview(myApp)
```

MATLAB opens a window showing the method signatures for `COM.Excel_Application` objects.

**Exploring 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 control or server and use the `eventlisteners` function to list registered events.

For example, to list the events for the Microsoft Internet Explorer web browser, type:

```matlab
myNet = actxserver('internetexplorer.application');
events(myNet)
```

MATLAB displays the events for your Internet Explorer version. To see which events have event handlers, type:

```matlab
eventlisteners(myNet)
```

```matlab
ans =
{}
```

An empty result means that no events are registered.
Exploring 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` or `actxcontrol` 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:

```matlab
e = actxserver('Excel.Application');
get(e)
```

MATLAB displays the properties, including interfaces, for your Excel version. For example, `Workbooks` is an interface.

```matlab
e.Workbooks
ans =
    Interface.000208DB_0000_0000_C000_000000000046
```

To explore the `Workbooks` interface, create a workbooks object and use the relevant MATLAB commands.

```matlab
w = e.Workbooks;
```

Identifying Objects and Interfaces

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class</code></td>
<td>Return the class of an object.</td>
</tr>
<tr>
<td><code>isa</code></td>
<td>Determine if an object is of a given MATLAB class.</td>
</tr>
<tr>
<td><code>iscom</code></td>
<td>Determine if the input is a COM or ActiveX object.</td>
</tr>
<tr>
<td><code>isevent</code></td>
<td>Determine if an item is an event of a COM object.</td>
</tr>
<tr>
<td><code>ismethod</code></td>
<td>Determine if an item is a method of a COM object.</td>
</tr>
<tr>
<td><code>isprop</code></td>
<td>Determine if an item is a property of a COM object.</td>
</tr>
<tr>
<td><code>isinterface</code></td>
<td>Determine if the input is a COM interface.</td>
</tr>
</tbody>
</table>

See Also

`eventlisteners` | `events` | `get` | `inspect` | `interfaces` | `methodsview`
More About

• “COM Object Properties” on page 12-21
• “Property Inspector” on page 12-22
• “COM Methods” on page 12-24
• “COM Events” on page 12-27
• “COM Event Handlers” on page 12-29

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.

```matlab
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.

```matlab
ws.StandardHeight
```

```matlab
ans =

14.4000
```

Display the function syntax for creating a Range object.

```matlab
methods(ws,'-full')
```

Search the displayed list for the Range entry: handle Range(handle, Variant, Variant(Optional))

Create a Range object consisting of the first row.

```matlab
wsRange = ws.Range('A1');
```

Increase the row height.

```matlab
wsRange.RowHeight = 25;
```

Open the worksheet, click in row one, and notice the height.

Close the workbook without saving.

```matlab
wb.Saved = 1;
```
Close(e.Workbook)

Close the application.

Quit(e)
delete(e)

See Also
methods

More About
• “Properties That Take Arguments” on page 12-23

External Websites
• Worksheet.Range Property (Excel) Office 2013
Write Data to Excel Spreadsheet Using ActiveX

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.

```matlab
e = actxserver('Excel.Application');
```

Add a workbook.

```matlab
eWorkbook = e.Workbooks.Add;
e.Visible = 1;
```

Make the first sheet active.

```matlab
eSheets = e.ActiveWorkbook.Sheets;
eSheet1 = eSheets.get('Item',1);
eSheet1.Activate;
```

Put MATLAB data into the worksheet.

```matlab
A = [1 2; 3 4];
eActivesheetRange = e.Activesheet.get('Range','A1:B2');
eActivesheetRange.Value = A;
```

Read the data back into MATLAB, where array `B` is a cell array.

```matlab
eRange = e.Activesheet.get('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.

```matlab
B = reshape([B{:}],size(B));
```

Save the workbook in a file.

```matlab
eWorkbook.SaveAs('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**
- “Import and Export Dates to Excel 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.

```matlab
h = actxserver('Excel.Application');
h.Visible = 1;
```

Open the Excel program and notice the cursor.

View the options for the `Cursor` property.

```matlab
set(h, 'Cursor')
```

```matlab
ans =
    'xlIBeam'
    'xlDefault'
    'xlNorthwestArrow'
    'xlWait'
```

These values are enumerated values, which means they are the only values allowed for the `Cursor` property.

Change the cursor to `'xlIBeam'`

```matlab
h.Cursor = 'xlIBeam';
```

Notice the cursor in the program.

Reset the cursor.

```matlab
h.Cursor = 'xlDefault';
```

```matlab
Quit(h)
delete(h)
```

Make sure that you close workbook objects you create to prevent potential memory leaks.

More About

• “Enumerated Values for Properties” on page 12-22
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

```vsto
e = actxserver('Excel.Application');
Add(e.Workbooks);
e.Visible = 1;
```

Create a collection of the three default sheets in the workbook.

```vsto
eSheets = e.ActiveWorkbook.Sheets;
```

Insert a sheet after the first item in the collection, `eSheet1`.

```vsto
eSheet1 = eSheets.Item(1);
eNewSheet = eSheets.Add([],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.

```vsto
Quit(e)
delete(e)
```

**More About**

- “Skipping Optional Input Arguments” on page 12-25

**External Websites**

- Office 2013 Sheets.Add Method (Excel)
Redraw Circle in \texttt{mwsamp} Control

This example shows how to call a method of the \texttt{mwsamp} control to redraw a circle.

Create the control.

\begin{verbatim}
myfigure = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200]);
\end{verbatim}

MATLAB opens a figure window and displays a circle and text label.

Display the properties of the control.

\begin{verbatim}
get(myfigure)
\end{verbatim}

\begin{verbatim}
Label: 'Label'
Radius: 20
Ret_IDispatch: [1x1 Interface.mwsamp2_ActiveX_Control_module._DMwsamp2]
\end{verbatim}

You can change the text displayed in the \texttt{Label} property and the radius of the circle.

Display the methods for the \texttt{mwsamp} control.

\begin{verbatim}
methods(myfigure)
\end{verbatim}

\textbf{Methods for class COM.mwsamp\textunderscore mwsampctrl\textunderscore 1:}

\begin{verbatim}
AboutBox GetR8Array SetR8 invoke
Beep GetR8Vector SetR8Array load
FireClickEvent GetVariantArray SetR8Vector move
GetBSTR GetVariantVector addproperty propedit
GetBSTRArray Redraw constructorargs release
GetI4 SetBSTR delete save
GetI4Array SetBSTRArray deleteproperty send
GetI4Vector SetI4 events set
GetIDispatch SetI4Array get
GetR8 SetI4Vector interfaces
\end{verbatim}

MATLAB displays the list alphabetically; method names with initial caps are listed before methods with lowercase names.

To use the \texttt{Redraw} method, display the method signatures which specify the input and output arguments.

\begin{verbatim}
methodsview(myfigure)
\end{verbatim}
MATLAB opens a window displaying method signatures for all methods supported by the object. The input argument for the Redraw method is the object handle.

Change the radius of circle `myfigure` to 100.

```
myfigure.Radius = 100;
Redraw(myfigure)
```

Close the figure window and the `methodsview` window.

**More About**
- “COM Methods” on page 12-24
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.

```matlab
excelapp = actxserver('Excel.Application');
wkbk = excelapp.Workbooks;
wdata = wkbk.Open('c:\work\weekly_log.xlsx');
```

Read data in the range D1 and F6 from sheet 2.

```matlab
sheets = wdata.Sheets;
sheet12 = sheets.Item(2);
range = sheet12.get('Range','D1','F6');
range.value
```

```matlab
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.

```matlab
function OnBeforeCloseWorkbook(varargin)
disp('BeforeClose event occurred')
end
```

Create the Excel object and make it visible.

```matlab
xl = actxserver('Excel.Application');
xl.Visible = 1;
```

Add a workbook.

```matlab
hWbks = xl.Workbooks;
hWorkbook = hWbks.Add;
```

Register the `OnBeforeCloseWorkbook` function for the **OnClose** event.

```matlab
registerevent(hWorkbook,{'BeforeClose' @OnBeforeCloseWorkbook})
```

Close the workbook, which triggers the **Close** event and calls the **OnClose** handler.

```matlab
Close(hWorkbook)
BeforeClose event occurred
```

**See Also**

registerevent

**More About**

- “COM Events” on page 12-27
Run Macro in Excel Server Application

In the following example, MATLAB runs the Microsoft Excel spreadsheet program in a COM server and invokes a macro that has been defined within the active spreadsheet file. The macro, `init_last`, takes no input parameters and is called from the MATLAB client using the statement:

```matlab
handle.ExecuteExcel4Macro('!macroname()');
```

Start the example by opening the spreadsheet file and recording a macro. The macro used here simply sets all items in the last column to zero. Save your changes to the spreadsheet.

Next, in MATLAB, create an Excel object, and open the spreadsheet:

```matlab
h = actxserver('Excel.Application');
wkbk = h.Workbooks;
file = wkbk.Open('d:\weatherlog\weekly.xls');
```

Open the sheet that you want to change, and retrieve the current values in the range of interest:

```matlab
sheets = file.Sheets;
sheet12 = sheets.Item('Week 12');
range = sheet12.get('Range', 'D1', 'F5');
range.Value
ans =
[ 78] [ 32] [ 37]
[ 69] [ 27] [ 30]
[ 77] [ 17] [ 16]
[ 74] [ -5] [ -1]
[ 68] [ 22] [ 35]
```

Now execute the macro, and verify that the values have changed as expected:

```matlab
h.ExecuteExcel4Macro('!init_last()');
range.Value
ans =
[ 78] [ 32] [ 0]
[ 69] [ 27] [ 0]
[ 77] [ 17] [ 0]
[ 74] [ -5] [ 0]
[ 68] [ 22] [ 0]
```
Combine Event Handlers as MATLAB Local Functions

This example shows how to consolidate event handlers into a single file using local functions.

Create the mycallbacks.m file containing three event handler routines, myclick, my2click, and mymoused, implemented as local functions.

```matlab
function a = mycallbacks(str)
    a = str2func(str);

function myclick(varargin)
    disp('Single click function')

function my2click(varargin)
    disp('Double click function')

function mymoused(varargin)
    disp('You have reached the mouse down function')
    disp('The X position is: ') double(varargin{5})
    disp('The Y position is: ') double(varargin{6})
```

The call to `str2func` converts the input string to a function handle.

Create the control.

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],gcf,'sampev')
```

Register Click event.

```matlab
registerevent(h,{'Click',mycallbacks('myclick')})
```

Related Examples

- “Display Event Messages from mwsamp Control” on page 12-55
MATLAB Sample ActiveX Control mwsamp

MATLAB includes an example COM control that draws a circle on the screen, displays some text, and fires events when the user single- or double-clicks the control.

Create the control by running the mwsamp function in the matlabroot\toolbox\matlab\winfun folder, or type:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 300 300]);
```

This control is in the folder with its type library. The type library is a binary file used by COM tools to decipher the capabilities of the control.

Display the events for this control.

```matlab
allEvents = events(h)
```

```matlab
allEvents =

    Click: 'void Click()'
    DblClick: 'void DblClick()'
    MouseDown: 'void MouseDown(int16 Button, int16 Shift, Variant x, Vari...'
    Event_Args: 'void Event_Args(int16 typeshort, int32 typelong, double t...'
```

allEvents is a structure array. Each field of the structure is the name of an event handler and the value of that field contains the signature for the handler routine. For example:

```matlab
allEvents_MouseDown
```

```matlab
ans =

    void MouseDown(int16 Button, int16 Shift, Variant x, Variant y)
```

See Also
actxcontrol

Related Examples
- “Display Event Messages from mwsamp Control” on page 12-55

More About
- “COM Event Handlers” on page 12-29
Display Event Messages from \texttt{mwsamp Control}

This example shows how to handle events fired by the MATLAB ActiveX control, \texttt{mwsamp2}.

Create Event Handler Routines

Create event handler files and save them to a folder on your path.

Create the \texttt{myclick.m} file.

\begin{verbatim}
function myclick(varargin)
disp('Single click function')
\end{verbatim}

Create the \texttt{my2click.m} file.

\begin{verbatim}
function my2click(varargin)
disp('Double click function')
\end{verbatim}

Create the \texttt{mymoused.m} file.

\begin{verbatim}
function mymoused(varargin)
disp('You have triggered the mouse down function')
disp('The X position is: ')
double(varargin{5})
disp('The Y position is: ')
double(varargin{6})
\end{verbatim}

Create Control and Register Events

Open a figure window.

\begin{verbatim}
f = figure('position', [100 200 200 200]);
\end{verbatim}

Register the Click and MouseDown events.

\begin{verbatim}
ctrl = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f,{'Click','myclick'; 'MouseDown','mymoused'});
\end{verbatim}

Respond to Control Events

When MATLAB creates the \texttt{mwsamp2} control, it also displays a figure window showing a label and circle at the center. If you click different positions in this window, you see a report in the MATLAB Command Window of the X and Y position of the mouse.
Each time you press the mouse button, the `MouseDown` event fires, calling the `mymoused` function. This function prints the position values for that event to the Command Window. For example:

```matlab
The X position is:
ans =
    [122]
The Y position is:
ans =
    [63]
```

The `Click` event displays the `myclick` message.

**Single click function**

Double-clicking the mouse does nothing, since the `DblClick` event is not registered.

**Register `DblClick` Event**

```matlab
registerevent(ctrl, {'DblClick', 'my2click'})
```

When you double-click the mouse, MATLAB displays both the `myclick` and `my2click` messages.

**Single click function**

**Double click function**

**Display Control Events**

Unregister the `DblClick` event.

```matlab
unregisterevent(ctrl, {'DblClick', 'my2click'})
```

Display the currently registered events.

```matlab
eventlisteners(ctrl)
```

```matlab
ans =
    'click'    'myclick'
    'mousedown'    'mymoused'
```

**Unregister All Events**

```matlab
unregisterallevents(ctrl)
eventlisteners(ctrl)
```
ans =
{
}

Clicking the mouse in the control window now does nothing since there are no active events.

Close the figure window.

**See Also**
actxcontrol | eventlisteners | registerevent | unregisterallevents

**Related Examples**
- “MATLAB Sample ActiveX Control mwsamp” on page 12-54

**More About**
- “COM Event Handlers” on page 12-29
Add Position Property to mwsamp Control

This example shows how to add a custom property, Position to the mwsamp control.

Create the control.

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2', [200 120 200 200]);
```

Create the property.

```matlab
addproperty(h,'Position')
```

Initialize the values.

```matlab
h.Position = [200 120];
```

List the properties of the control.

```matlab
get(h)
```

- **Label**: 'Label'
- **Radius**: 20
- **Ret_IDispatch**: [1x1 Interface.mwsamp2_ActiveX_Control_module._DMwsamp2]
- **Position**: [200 120]

Property Position has been added.

More About

- “Custom Properties” on page 12-23
Save mwsamp2 COM Control

This example shows how to restore original mwsamp2 control settings.

Create the mwsamp2 control.

```matlab
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
```

Save the control settings to a new file, mwsample.

```matlab
save(h, 'mwsample')
```

Alter the figure by changing its label and the radius of the circle.

```matlab
h.Label = 'Circle';
h.Radius = 50;
Redraw(h)
```

Restore the control to its original state.

```matlab
load(h, 'mwsample')
```

Verify the results.

```matlab
get(h)
```

```
  Label: 'Label'
  Radius: 20
  Ret_IDispatch: [1x1 Interface.mwsamp2_ActiveX_Control_module._DMwsamp2]
```

MATLAB displays the original settings.
Deploy ActiveX Controls Requiring Run-Time Licenses

In this section...

“Create a Function to Build the Control” on page 12-60
“Build the Control and the License File” on page 12-60
“Build the Executable” on page 12-61
“Deploy the Files” on page 12-61

When you deploy a Microsoft ActiveX control that requires a run-time license, include a license key, which the control reads at run-time. If the key matches the control's version of the license key, an instance of the control is created. Use the following procedure to deploy a run-time-licensed control with a MATLAB application.

Create a Function to Build the Control

First, create a function to build the control and save is as a .m file. The file must contain two elements:

- The pragma %#function actxlicense. This pragma causes the MATLAB Compiler to embed a function named actxlicense into the standalone executable file you build.
- A call to actxcontrol to create an instance of the control.

Place this file in a folder outside of the MATLAB code tree.

Here is an example file:

```matlab
function buildcontrol
  %#function actxlicense
  h=actxcontrol('MFCCONTROL2.MFCCControl2Ctrl.1',[10 10 200 200]);
```

Build the Control and the License File

Change to the folder where you placed the function you created to build the control. Call the function. When it executes this function, MATLAB determines whether the control requires a run-time license. If it does, MATLAB creates another file, named actxlicense.m, in the current working folder. The actxlicense function defined in this file provides the license key to MATLAB at run-time.
Build the Executable

Next, call MATLAB Compiler to create the standalone executable from the file you created to build the control. The executable contains both the function that builds the control and the `actxlicense` function.

`mcc -m buildcontrol`

Deploy the Files

Finally, distribute `buildcontrol.exe`, `buildcontrol.ctf`, and the control (.ocx or .dll).
Microsoft Forms 2.0 Controls

In this section...

“AFFECTED CONTROLS” on page 12-62
“REPLACEMENT CONTROLS” on page 12-62

Microsoft Forms 2.0 controls are designed for use with applications enabled by Microsoft Visual Basic for Applications (VBA). An example is Microsoft Office software.

If you encounter problems when creating or using Forms 2.0 controls, use the following replacement controls, or consult article 236458 in the Microsoft Knowledge Base for further information:


Affected Controls

- Forms.TextBox.1
- Forms.CheckBox.1
- Forms.CommandButton.1
- Forms.Image.1
- Forms.OptionButton.1
- Forms.ScrollBar.1
- Forms.SpinButton.1
- Forms.TabPage.1
- Forms.ToggleButton.1

Replacement Controls

Microsoft recommends the following replacements:

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forms.TextBox.1</td>
<td>RICHTEXT.RichtextCtrl.1</td>
</tr>
<tr>
<td>Forms.CheckBox.1</td>
<td>vidtc3.Checkbox</td>
</tr>
<tr>
<td>Old</td>
<td>New</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>Forms.CommandButton.1</td>
<td>MSComCtl2.FlatScrollBar.2</td>
</tr>
<tr>
<td>Forms.TabStrip.1</td>
<td>COMCTL.TabStrip.1</td>
</tr>
</tbody>
</table>
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 control or server that supports the collection. Although integer indices are common, the index could also be a string 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:

```matlab
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 vice-versa. 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 (IUnknown) with 64-bit MATLAB.
• MATLAB only supports indexed collections.
• COM controls are not printed with figure windows.
• “Unsupported Types” on page 12-15
• MATLAB does not support asynchronous events.
• A MATLAB COM ActiveX control container does not in-place activate controls until they are visible.
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:

\[
\text{handle.PutFullMatrix('a','base',7,[5 8])};
\]

MATLAB displays:

??? Error: Type mismatch, argument 3.

In this case, the argument, 7, is invalid because \text{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:

\[
\text{PutFullMatrix(handle,'a','base',7,[5 8])};
\]

MATLAB displays:

??? Error: Type mismatch, argument 3.

In this call to the \text{PutFullMatrix} function, 7 is argument four. However, the COM server does not receive the first argument. The \text{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.

More About

- “COM Methods” on page 12-24
MATLAB COM Automation Server Support

- “Register MATLAB as Automation Server” on page 13-2
- “MATLAB COM Automation Server Interface” on page 13-4
- “Create MATLAB Server” on page 13-7
- “Connect to Existing MATLAB Server” on page 13-9
- “MATLAB Application as DCOM Server” on page 13-10
- “VT_DATE Data Type” on page 13-11
- “Data Types For Visual Basic .NET Clients” on page 13-12
- “Visible Property” on page 13-13
- “Shared or Dedicated Server” on page 13-14
- “Manually Create Automation Server” on page 13-15
- “Launch MATLAB as Automation Server in Desktop Mode” on page 13-16
- “Call MATLAB Function from Visual Basic .NET Client” on page 13-17
- “Pass Complex Data to MATLAB from C# Client” on page 13-18
- “Call MATLAB Function from C# Client” on page 13-20
- “View MATLAB Functions from Visual Basic .NET Object Browser” on page 13-22
- “Waiting for MATLAB Application to Complete” on page 13-23
- “Conversion of MATLAB Types to COM Types” on page 13-24
- “Conversion of COM Types to MATLAB Types” on page 13-26
Register MATLAB as Automation Server

In this section...

| “When to Register MATLAB” on page 13-2 |
| “Register from System Prompt” on page 13-2 |
| “Register from MATLAB Command Prompt” on page 13-3 |

When to Register MATLAB

To use MATLAB as an Automation server, you first must register the application in the Windows registry, which happens when you install a new version of MATLAB. If you have multiple versions of MATLAB installed on your system, only one version is registered. By default, this version is the last version you installed. This version remains registered until you manually change the registry, or install a new version.

To manually register MATLAB, choose one of these options:

- “Register from System Prompt” on page 13-2
- “Register from MATLAB Command Prompt” on page 13-3

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. Because you open MATLAB to run this command, you know what MATLAB version you will register.

!matlab -regserver

MATLAB opens a minimized command window. Open this window and exit MATLAB.

**See Also**
matlab (Windows)

**More About**

- “Create MATLAB Server” on page 13-7
MATLAB COM Automation Server Interface

In this section...

“COM Server Types” on page 13-4
“Shared and Dedicated Servers” on page 13-4
“Programmatic Identifiers” on page 13-4
“In-Process and Out-of-Process Servers” on page 13-5

COM Server Types

There are three types of COM servers:

• 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. Custom servers are preferred when faster client access is critical.

• Dual — A server that implements a combination of Automation and Custom interfaces.

Shared and Dedicated Servers

The MATLAB Automation server has two modes:

• Shared — One or more client applications connect to the same MATLAB server. All clients share server.

• Dedicated — Each client application creates its own dedicated MATLAB server.

If you use `matlab.application` as your ProgID, MATLAB creates a shared server. For more information, see “Shared or Dedicated Server” on page 13-14.

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).

### In-Process and Out-of-Process Servers

You can configure a server three ways. MATLAB supports all of these configurations.

- “In-Process Server” on page 13-5
- “Remote Out-of-Process Server” on page 13-6

### In-Process Server

An in-process server is a component implemented as a dynamic link library (DLL) or ActiveX control 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

This is a type of out-of-process server; however, the client and server processes are on different systems and communicate over a network. Network communications, in addition to the overhead required for data transfer, can make this configuration slower than the local out-of-process configuration. This configuration runs only on systems that support the Distributed Component Object Model (DCOM).
Create MATLAB Server

In this section...
“Getting Started” on page 13-7
“Get or Set the Status of a MATLAB Automation Server” on page 13-8

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 13-4.

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.
  ```vb
  MatLab = CreateObject("Matlab.Application")
  ```
- 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:

```vb
Matlab = New MLApp.MLApp
```

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:

```matlab
enableservice('AutomationServer',true)
```

To determine the current state of a MATLAB Automation server, type:

```matlab
enableservice('AutomationServer')
```

If MATLAB displays:

```matlab
ans =
    1
```

then MATLAB is currently an Automation server.

See Also

`enableservice`

Related Examples

• “Manually Create Automation Server” on page 13-15

More About

• “Register MATLAB as Automation Server” on page 13-2
• “Programmatic Identifiers” on page 13-4
• “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:

```vbnet
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 13-7.

```vbnet
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])")
```
MATLAB Application as DCOM 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, you can use the controller to start a MATLAB server on a remote machine.

To do this, 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 may not be running 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 the DCOM documentation for how to configure DCOM for your environment.

External Websites

VT_DATE Data Type

To pass a VT_DATE type input to a Visual Basic program or an ActiveX control method, use the MATLAB class COM.date. For example:

```matlab
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:

```matlab
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).
Shared or Dedicated Server

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.

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.
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\R2006a\bin\win32\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.
Launch MATLAB as Automation Server in Desktop Mode

To launch MATLAB as a COM Automation server in full desktop mode, use the programmatic identifier `Matlab.Desktop.Application`. For example, in Microsoft Visual Basic .NET:

```vbscript
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 a user-defined MATLAB function named `solve_bvp` from a Microsoft Visual Basic client application through a COM interface. It also plots a graph in a new MATLAB window and performs a simple computation:

```vbc
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag(1, 3) As Double

MatLab = CreateObject("Matlab.Application")

'Calling MATLAB function from VB
'Assuming solve_bvp exists at specified location
Result = MatLab.Execute("cd d:\matlab\work\bvp")
Result = MatLab.Execute("solve_bvp")

'Executing other MATLAB commands
Result = MatLab.Execute("surf(peaks)")
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8]\n")
Result = MatLab.Execute("b = a + a")
'Bring matrix b into VB program
MatLab.GetFullMatrix("b", "base", MReal, MImag)
```
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:

```csharp
MLApp.MLApp matlab = new MLApp.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.

Here is the complete example:

```csharp
using System;
namespace ConsoleApplication4
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            MLApp.MLApp matlab = new MLApp.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`.

```matlab
function [x,y] = myfunc(a,b,c)
x = a + b;
y = sprintf('Hello %s',c);
```

Create the C# application.

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create the MATLAB instance
            MLApp.MLApp matlab = new MLApp.MLApp();

            // 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();
        }
    }
}
```
In Microsoft Visual Studio, add a reference to your C# project to the MATLAB COM object. From the Project menu, select Add Reference.

Select the COM tab in the Add Reference dialog box.

Select the MATLAB application.
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:

```vbnet
Dim Matlab As MLApp.MLApp
Private Sub View_Methods()
    Matlab = New MLApp.MLApp
    '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.

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?
• Server Busy: Matlab called from Visual Basic (VB, VBA)
Conversion of MATLAB Types to COM Types

The following table shows how MATLAB converts data from MATLAB to COM types.

<table>
<thead>
<tr>
<th>MATLAB Type</th>
<th>Closest COM Type</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>VT_DISPATCH</td>
<td>VT_DISPATCH</td>
</tr>
<tr>
<td></td>
<td>VT_UNKNOWN</td>
<td>VT_UNKNOWN</td>
</tr>
<tr>
<td>string</td>
<td>VT_BSTR</td>
<td>VT_LPWSTR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_LPSTR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_BSTR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_FILETIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_ERROR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_DECIMAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_CLSID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_DATE</td>
</tr>
<tr>
<td>int16</td>
<td>VT_I2</td>
<td>VT_I2</td>
</tr>
<tr>
<td>uint16</td>
<td>VT_UI2</td>
<td>VT_UI2</td>
</tr>
<tr>
<td>int32</td>
<td>VT_I4</td>
<td>VT_I4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_INT</td>
</tr>
<tr>
<td>uint32</td>
<td>VT_UI4</td>
<td>VT_UI4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_UINT</td>
</tr>
<tr>
<td>int64</td>
<td>VT_I8</td>
<td>VT_I8</td>
</tr>
<tr>
<td>uint64</td>
<td>VT_UI8</td>
<td>VT_UI8</td>
</tr>
<tr>
<td>single</td>
<td>VT_R4</td>
<td>VT_R4</td>
</tr>
<tr>
<td>double</td>
<td>VT_R8</td>
<td>VT_R8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_CY</td>
</tr>
<tr>
<td>logical</td>
<td>VT_BOOL</td>
<td>VT_BOOL</td>
</tr>
<tr>
<td>char</td>
<td>VT_I1</td>
<td>VT_I1</td>
</tr>
</tbody>
</table>
|             |                 | 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.

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Closest COM Type</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>variant</td>
<td>VT_VARIANT</td>
<td>VT_VARIANT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_USERDEFINED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VT_ARRAY</td>
</tr>
<tr>
<td>variant(pointer)</td>
<td>VT_VARIANT</td>
<td>VT_VARIANT</td>
</tr>
</tbody>
</table>

SAFEARRAY Data

When a COM method identifies a `SAFEARRAY` or `SAFEARRAY(pointer)`, the MATLAB equivalent is a matrix.

<table>
<thead>
<tr>
<th>MATLAB Argument</th>
<th>Closest COM Type</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAFEARRAY</td>
<td>VTSAFEARRAY</td>
<td>VTSAFEARRAY</td>
</tr>
<tr>
<td>SAFEARRAY(pointer)</td>
<td>VTSAFEARRAY</td>
<td>VTSAFEARRAY</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>COM Variant Type</th>
<th>Description</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_DISPATCH</td>
<td>IDispatch *</td>
<td>handle</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>IUnknown *</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATLAB does not support the IUnknown and IDispatch interfaces with 64-bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATLAB.</td>
<td></td>
</tr>
<tr>
<td>VT_LPWSTR</td>
<td>wide null terminated string</td>
<td>string</td>
</tr>
<tr>
<td>VT_LPSTR</td>
<td>null terminated string</td>
<td></td>
</tr>
<tr>
<td>VT_BSTR</td>
<td>OLE Automation string</td>
<td></td>
</tr>
<tr>
<td>VT_FILETIME</td>
<td>FILETIME</td>
<td></td>
</tr>
<tr>
<td>VT_ERROR</td>
<td>SCODE</td>
<td></td>
</tr>
<tr>
<td>VT_DECIMAL</td>
<td>16-byte fixed point</td>
<td></td>
</tr>
<tr>
<td>VT_CLSID</td>
<td>Class ID</td>
<td></td>
</tr>
<tr>
<td>VT_DATE</td>
<td>date</td>
<td></td>
</tr>
<tr>
<td>VT_INT</td>
<td>signed machine int</td>
<td>double</td>
</tr>
<tr>
<td>VT_UINT</td>
<td>unsigned machine int</td>
<td></td>
</tr>
<tr>
<td>VT_I2</td>
<td>2 byte signed int</td>
<td></td>
</tr>
<tr>
<td>VT_UI2</td>
<td>unsigned short</td>
<td></td>
</tr>
<tr>
<td>VT_I4</td>
<td>4 byte signed int</td>
<td></td>
</tr>
<tr>
<td>VT_UI4</td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>VT_R4</td>
<td>4 byte real</td>
<td></td>
</tr>
<tr>
<td>VT_R8</td>
<td>8 byte real</td>
<td></td>
</tr>
<tr>
<td>VT_CY</td>
<td>currency</td>
<td></td>
</tr>
<tr>
<td>VT_I8</td>
<td>signed int64</td>
<td>int64</td>
</tr>
<tr>
<td>VT_UI8</td>
<td>unsigned int64</td>
<td>uint64</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td></td>
<td>logical</td>
</tr>
<tr>
<td>VT_I1</td>
<td>signed char</td>
<td>char</td>
</tr>
<tr>
<td>VT_UI1</td>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>VARIANT *</td>
<td>variant</td>
</tr>
</tbody>
</table>
### COM Variant Type

<table>
<thead>
<tr>
<th>COM Variant Type</th>
<th>Description</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_USERDEFINED</td>
<td>user-defined type</td>
<td>SAFEARRAY*</td>
</tr>
<tr>
<td>VT_ARRAY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>VT_BYREF</td>
<td>VARIANT *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void* for local use</td>
</tr>
<tr>
<td>VTSAFEARRAY</td>
<td>use VT_ARRAY in VARIANT</td>
<td>SAFEARRAY</td>
</tr>
<tr>
<td>VTSAFEARRAY</td>
<td>VT_BYREF</td>
<td>SAFEARRAY(pointer)</td>
</tr>
</tbody>
</table>

### See Also

PutWorkspaceData
Using Web Services with MATLAB

• “Set Up WSDL Tools” on page 14-2
• “Display a World Map” on page 14-3
• “Using WSDL Web Service with MATLAB” on page 14-8
• “Access Services That Use WSDL Documents” on page 14-10
• “Error Handling” on page 14-12
• “XML-MATLAB Data Type Conversion” on page 14-14
• “Limitations to WSDL Document Support” on page 14-16
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 Oracle® Java JDK and the Apache™ CXF programs. While the programs are available at no charge, they require several hundred megabytes of disk space.

- Download and install the JDK software from the Java SE Downloads Web page. Use Java SE version 7uXX, where XX is the latest update.
- Download and install Apache CXF software from the Apache CXF Web page. Choose the latest release of version 2.7.
- Make note of the installation folders created by these programs. Set the paths to these variables, jdk = 'YOUR_JDK_PATH' and cxf = 'YOUR_CXF_PATH', then call:

  matlab.wsd1.setWSDLToolPath('JDK',jdk,'CXF',cxf)

See Also
matlab.wsd1.setWSDLToolPath

External Websites
- http://www.oracle.com/technetwork/java/javase/downloads
- http://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.

You must install the Java JDK and Apache CXF programs and set the tool paths to run this example.

```matlab
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.

```matlab
wsdlFile = ...
'http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapServer?wsdl';
```

Create the class files for the client.

```matlab
matlab.wsdl.createWSDLClient(wsdlFile)
```

Created USGSImageryOnly_MapServer.

```
."\USGSImageryOnly_MapServer.m
."+wsdl
```

In order to use USGSImageryOnly_MapServer, you must run javaaddpath('."+wsdl\mapserver')

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/MapServer and returns a SERVICE.

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 web client
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 codes 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 a 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)

Related Examples
• “Set Up WSDL Tools” on page 14-2
Using WSDL Web Service with MATLAB

In this section...

“What Are Web Services in MATLAB?” on page 14-8
“What are WSDL Documents?” on page 14-8
“What You Need to Use WSDL with MATLAB” on page 14-9

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.createWSDLClient` 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”.
Access Services That Use WSDL Documents

A WSDL document uses a standard format to describe a server's operations, arguments, and transactions. The \texttt{matlab.wsdl.createWSDLClient} function creates a MATLAB class that allows you to use the server APIs.

To use the \texttt{matlab.wsdl.createWSDLClient} function, you need to know the location of the service's 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. \texttt{matlab.wsdl.createWSDLClient} does not support RPC-encoded.

You must download supported versions of the 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 \texttt{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 \texttt{matlab.wsdl.createWSDLClient}, supplying the WSDL document location, which can be a URL or a path to a file.

The function converts the server's APIs to a MATLAB class, and creates a class folder in the current folder. The class folder contains methods for using the server's APIs. The function always creates a constructor method that has the same name as the class.

You only need to run the \texttt{matlab.wsdl.createWSDLClient} function once. You can access the class anytime after that.

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 vice versa.

**See Also**
matlab.wsd1.createWSDLClient | matlab.wsd1.setWSDLToolPath

**Related Examples**
- “Set Up WSDL Tools” on page 14-2

**More About**
- “XML-MATLAB Data Type Conversion” on page 14-14
- “Limitations to WSDL Document Support” on page 14-16
Error Handling

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Considerations Using Web Services” on page 14-12</td>
</tr>
<tr>
<td>“Error Handling with try/catch Statements” on page 14-12</td>
</tr>
<tr>
<td>“Use a Local Copy of the WSDL Document” on page 14-12</td>
</tr>
<tr>
<td>“Java Errors Accessing Service” on page 14-13</td>
</tr>
<tr>
<td>“Anonymous Types Not Supported” on page 14-13</td>
</tr>
</tbody>
</table>

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.createWSDLClient` 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.

```matlab
wsdlURL = ...;
wsdlFile = 'USGSImageryOnly_MapServer';
if ~exist(wsdlFile,'file')
```

14-12
websave(wsd1File, wsdlURL)
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 `matlab1.wsdl.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 `matlab1.wsdl.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, “Web services hints and tips: avoid anonymous types,” at http://www.ibm.com/developerworks/library/ws-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 MATLAB types (classes), and vice-versa. The following table contains the XML type and the corresponding MATLAB type for scalar values used in a WSDL document.

<table>
<thead>
<tr>
<th>XML Schema Type</th>
<th>MATLAB Type Returned—Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>logical</td>
</tr>
<tr>
<td>byte</td>
<td>int8</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>uint8</td>
</tr>
<tr>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>uint16</td>
</tr>
<tr>
<td>int</td>
<td>int32</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>uint32</td>
</tr>
<tr>
<td>long</td>
<td>int64</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>uint64</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>string</td>
<td>char array</td>
</tr>
<tr>
<td>gYear, gMonth, gDay, gYearMonth, gMonthDay</td>
<td>calendarDuration array</td>
</tr>
<tr>
<td>dateTime</td>
<td>dateTime</td>
</tr>
<tr>
<td>date</td>
<td>dateTime with Year, Month, Day fields undefined.</td>
</tr>
<tr>
<td>time</td>
<td>dateTime with Hours, Minutes, Seconds fields undefined.</td>
</tr>
<tr>
<td>duration</td>
<td>duration if no year, month or day calendarDuration otherwise</td>
</tr>
<tr>
<td>NOTATION, QName</td>
<td>string containing a legal QName</td>
</tr>
<tr>
<td>hexbinary, base64Binary</td>
<td>N-by-1 vector of uint8 representing byte values (0-255) of encoded data</td>
</tr>
</tbody>
</table>
### XML Schema Type

<table>
<thead>
<tr>
<th>XML Schema Type</th>
<th>MATLAB Type Returned—Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal, integer, nonPositiveInteger, nonNegativeInteger, positiveInteger, negativeInteger</td>
<td>double array</td>
</tr>
</tbody>
</table>

The following table contains the XML type and the corresponding MATLAB type for arrays.

### XML Schema Type—Array

<table>
<thead>
<tr>
<th>XML Schema Type—Array</th>
<th>MATLAB Type Returned—Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>array of string</td>
<td>N-by-1 cell array of string</td>
</tr>
<tr>
<td>array of any type</td>
<td>N-by-1 vector of specified type</td>
</tr>
<tr>
<td>array of hexbinary, base64Binary</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
Limitations to WSDL Document Support

RPC-encoded WSDL documents are not supported. For these documents, use `createClassFromWsdl`.

The following WSDL documents are not supported:

- 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.

```xml
<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.

```xml
<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 file.
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.

```xml
<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.

```xml
<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.

  ```
  Or:
  Error using matlab.internal.callJava
  No GetMyOp method with appropriate signature exists in Java class $Proxy57
  ```
System Commands

- “Shell Escape Functions” on page 15-2
- “Run External Commands, Scripts, and Programs” on page 15-3
- “Change Environment Variable for Shell Command” on page 15-6
Shell Escape Functions

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.

```matlab
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.

Related Examples

- “Run External Commands, Scripts, and Programs” on page 15-3
Run External Commands, Scripts, and Programs

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. Use it to invoke utilities or call other executable programs without quitting MATLAB. To use the exclamation point in a factorial expression, call the factorial function.

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. Add & to the end of the line, such as

!dir &

on Windows platforms to display the output in a separate window or to run the application in background mode. For example:

!excel.exe &

opens Microsoft Excel software and returns control to the Command Window so you can continue running MATLAB language statements.

Restrictions maintained within the operating system determine the maximum length of the argument list you can provide as input to the bang (!) command.

For details about running external programs that return results and status, see the unix, dos, and system functions.

Note To execute operating system commands with specific environment variables, include all commands to the operating system within the system call. Separate the commands using & (ampersand) for DOS, and ; (semicolon) for UNIX platforms. This applies to the MATLAB ! (bang), dos, unix, and system functions. Another approach is to set environment variables before starting MATLAB.

On Mac platforms, you cannot run AppleScript (from Apple) directly from MATLAB. To run AppleScript commands, call the Apple Mac OS X osascript function using the MATLAB unix or ! (bang) functions.
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 determine the system path that is visible to MATLAB, type the following in the Command Window:

```
getenv('PATH')
```

You can make modifications to the system path that persist for the current MATLAB session or across subsequent MATLAB sessions, as described in the sections that follow.

Modify System Path for Current MATLAB Session

Do one of the following:

- Change the current folder in MATLAB to the folder that contains the program you want to run.
- Issue these commands using the Command Window:

```
path1 = getenv('PATH')
path1 = [path1 ':/usr/local/bin']
setenv('PATH', path1)
!echo $PATH
```

If you restart MATLAB, the folder is no longer on the system path visible to MATLAB.

Modify System Path Across MATLAB Sessions Within Current Shell Session

To add a folder to the system path from the shell:

1. Stop 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:
   - If you are using `bash` or a related shell, type:

     ```
     export PATH="$PATH:myfolder"
     ```
   - If you are using `tcsh` or a related shell, type:

     ```
     setenv PATH "${PATH}:myfolder"
     ```
3. Start MATLAB.
4. In the MATLAB Command Window, type:
!echo $PATH

If you restart MATLAB within the current shell session, the folder remains on the system path visible to MATLAB. However, if you restart the shell session, and then restart MATLAB, the folder is no longer on the system path visible to MATLAB.

**Modify System 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”.

```matlab
path1 = getenv('PATH')
path1 = [path1 ':/usr/local/bin']
setenv('PATH', path1)
!echo $PATH
```

**See Also**

dos | factorial | system | unix
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.

```bash
#!/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`
Serial Port I/O

- “Capabilities and Supported Interfaces and Platforms” on page 16-2
- “Overview of the Serial Port” on page 16-4
- “Getting Started with Serial I/O” on page 16-16
- “Create a Serial Port Object” on page 16-22
- “Connect to the Device” on page 16-26
- “Configure Communication Settings” on page 16-27
- “Write and Read Data” on page 16-28
- “Events and Callbacks” on page 16-46
- “Control Pins” on page 16-54
- “Debugging: Recording Information to Disk” on page 16-60
- “Save and Load” on page 16-66
- “Disconnect and Clean Up” on page 16-68
- “Property Reference” on page 16-70
- “Properties — Alphabetical List” on page 16-74
Capabilities and Supported Interfaces and Platforms

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“What Is the MATLAB Serial Port Interface?” on page 16-2</td>
</tr>
<tr>
<td>“Supported Serial Port Interface Standards” on page 16-3</td>
</tr>
<tr>
<td>“Supported Platforms” on page 16-3</td>
</tr>
<tr>
<td>“Using the Examples with Your Device” on page 16-3</td>
</tr>
</tbody>
</table>

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 http://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 16-4. 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 32-bit
- Linux 64-bit
- Mac OS X
- Mac OS X 64-bit
- Microsoft Windows 32-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.
Overview of the Serial Port

In this section...

“Introduction” on page 16-4
“What Is Serial Communication?” on page 16-4
“The Serial Port Interface Standard” on page 16-4
“Connecting Two Devices with a Serial Cable” on page 16-5
“Serial Port Signals and Pin Assignments” on page 16-6
“Serial Data Format” on page 16-10
“Finding Serial Port Information for Your Platform” on page 16-13
“Using Virtual USB Serial Ports” on page 16-15
“Selected Bibliography” on page 16-15

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 16-16 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 16-6.
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**
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.

![Diagram showing on and off states for data and control signals.]

**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 16-8
- “The DTR and DSR Pins” on page 16-9
- “The CD and RI Pins” on page 16-9

**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 16-57.

**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.

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.

**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.

![Serial Data Format Diagram]

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 16-28.

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 16-29.

**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 16-10, 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**

<table>
<thead>
<tr>
<th>Parity Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even</td>
<td>The data bits plus the parity bit result in an even number of 1s.</td>
</tr>
<tr>
<td>Mark</td>
<td>The parity bit is always 1.</td>
</tr>
<tr>
<td>Odd</td>
<td>The data bits plus the parity bit result in an odd number of 1s.</td>
</tr>
<tr>
<td>Space</td>
<td>The parity bit is always 0.</td>
</tr>
</tbody>
</table>

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.

**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.

MATLAB displays the following Ports dialog box.
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: infinite
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 `/dev/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” 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**


Getting Started with Serial I/O

In this section...

“Example: Getting Started” on page 16-16
“The Serial Port Session” on page 16-16
“Configuring and Returning Properties” on page 16-17

Example: Getting Started

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.

```matlab
s = serial('COM1');
set(s,'BaudRate',4800);
fopen(s);
fprintf(s,'*IDN?
out = fscanf(s);
fclose(s)
delete(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. Create a serial port object — Create a serial port object for a specific serial port using the `serial` 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.

2. Connect to the device — Connect the serial port object to the device using the `fopen` function.
   
   After the object is connected, 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 the `set` function or 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 `fprintf` or `fwrite` function, and read data from the device using the `fgetl`, `fgets`, `fread`, `fscanf`, or `readasync` 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, disconnect it from the device using the `fclose` function, remove it from memory using the `delete` function, and 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 “Example: Getting Started” on page 16-16.

**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.

```matlab
s = serial('COM1');
set(s)
```

```plaintext
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:

```matlab
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`.

```matlab
get(s,'OutputBufferSize')
ans =
    512
```

To display the current values for multiple properties, include the property names as elements of a cell array.

```matlab
get(s,{'Parity','TransferStatus'})
ans =
    'none'    'idle'
```

Use the dot notation to display a single property value.

```matlab
s.Parity
ans =
    none
```

### Configuring Property Values

You can configure property values using the `set` function:

```matlab
set(s,'BaudRate',4800);
```

or the dot notation:

```matlab
s.BaudRate = 4800;
```

To configure values for multiple properties, supply multiple property name/property value pairs to `set`.

```matlab
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 16-70.

### 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**.

```matlab
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

You can find the default value for any property in the property reference pages.
Create a Serial Port Object

In this section...

“Overview of a Serial Port Object” on page 16-22
“Configuring Properties During Object Creation” on page 16-23
“The Serial Port Object Display” on page 16-24
“Creating an Array of Serial Port Objects” on page 16-24

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:

```matlab
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 Instrument Control Toolbox function

```matlab
instrhwinfo('serial')
```

provides a list of available serial ports. This list is an example of serial constructors on different platforms:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Serial Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 32 and 64-bit</td>
<td><code>serial('/dev/ttyS0');</code></td>
</tr>
<tr>
<td>Mac OS X</td>
<td><code>serial('/dev/tty.KeySerial1&gt;');</code></td>
</tr>
<tr>
<td>and Mac OS X 64-bit</td>
<td></td>
</tr>
<tr>
<td>Microsoft Windows 32 and 64-bit</td>
<td><code>serial('com1');</code></td>
</tr>
<tr>
<td>Sun™ Solaris™ 64-bit</td>
<td><code>serial('/dev/term/a');</code></td>
</tr>
</tbody>
</table>

The serial port object `s` now exists in the MATLAB workspace. To display the class of `s`, use the `whos` command.

```matlab
whos s
Name      Size         Bytes  Class
16-22
Create a Serial Port Object

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**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the serial port object</td>
</tr>
<tr>
<td>Port</td>
<td>Indicate the platform-specific serial port name</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type</td>
</tr>
</tbody>
</table>

Display the values of these properties for `s` with the `get` function. On a Windows platform, it will look like this:

```matlab
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.

```matlab
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');
```
Create a Serial Port Object

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Status</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>serial</td>
<td>closed</td>
<td>Serial-COM1</td>
</tr>
<tr>
<td>2</td>
<td>serial</td>
<td>closed</td>
<td>Serial-COM2</td>
</tr>
</tbody>
</table>

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 16-70.

**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. The following table describes these properties.

**Communication Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaudRate</td>
<td>Specify the rate at which bits are transmitted</td>
</tr>
<tr>
<td>DataBits</td>
<td>Specify the number of data bits to transmit</td>
</tr>
<tr>
<td>Parity</td>
<td>Specify the type of parity checking</td>
</tr>
<tr>
<td>StopBits</td>
<td>Specify the number of bits used to indicate the end of a byte</td>
</tr>
<tr>
<td>Terminator</td>
<td>Specify the terminator character</td>
</tr>
</tbody>
</table>

**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 16-28
“Example — Introduction to Writing and Reading Data” on page 16-28
“Controlling Access to the MATLAB Command Line” on page 16-29
“Writing Data” on page 16-30
“Reading Data” on page 16-35
“Example — Writing and Reading Text Data” on page 16-41
“Example — Parsing Input Data Using textscan” on page 16-42
“Example — Reading Binary Data” on page 16-43

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 16-30. For read operations, these questions are answered in “Reading Data” on page 16-35.

Note: All the examples shown below are based on a Windows 32-bit platform. Refer to “Overview of a Serial Port Object” on page 16-22 section for information about other platforms.

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 16-10. 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 16-46).

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.

![Diagram of serial port communication](image)

**Writing Data**

This section describes writing data to your serial port device in three parts:

- “The Output Buffer and Data Flow” on page 16-31 describes the flow of data from MATLAB software to the device.
- “Writing Text Data” on page 16-32 describes how to write text data (string commands) to the device.
- “Writing Binary Data” on page 16-34 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**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fprintf</td>
<td>Write text to the device</td>
</tr>
<tr>
<td>fwrite</td>
<td>Write binary data to the device</td>
</tr>
</tbody>
</table>
Write and Read Data

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations</td>
</tr>
</tbody>
</table>

The following table shows the properties associated with writing data.

**Properties Associated with Writing Data**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesToOutput</td>
<td>Number of bytes currently in the output buffer</td>
</tr>
<tr>
<td>OutputBufferSize</td>
<td>Size of the output buffer in bytes</td>
</tr>
<tr>
<td>Timeout</td>
<td>Waiting time to complete a read or write operation</td>
</tr>
<tr>
<td>TransferStatus</td>
<td>Indicate if an asynchronous read or write operation is in progress</td>
</tr>
<tr>
<td>ValuesSent</td>
<td>Total number of values written to the device</td>
</tr>
</tbody>
</table>

**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:

```matlab
s = serial('COM1');
get(s,{'OutputBufferSize','BytesToOutput'})
an = [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.
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 16-10. fprintf and the terminator are discussed in “Writing Text Data” on page 16-32.

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.

### 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.

```matlab
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.

```matlab
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`.

```matlab
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`.

```matlab
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 16-32. 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.
<table>
<thead>
<tr>
<th>Error</th>
<th>Occurs when</th>
<th>Troubleshooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>??? Error using ==&gt; serial.fwrite at 199 OBJ must be connected to the hardware with FOPEN.</td>
<td>You perform a write operation and the serial port object is not connected to the device.</td>
<td>Use fopen to establish a connection to the device.</td>
</tr>
<tr>
<td>??? Error using ==&gt; serial.fwrite at 199 The number of bytes written must be less than or equal to OutputBufferSize-BytesToOutput.</td>
<td>The output buffer is not able to hold all the data to be written.</td>
<td>Specify the size of the output buffer with the OutputBufferSize property.</td>
</tr>
<tr>
<td>??? Error using ==&gt; serial.fwrite at 199 FWRITE cannot be called. The FlowControl 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.</td>
<td>• You set the flowcontrol 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.</td>
<td>Check your remote device status and flow control settings to see if hardware flow control is causing MATLAB errors.</td>
</tr>
</tbody>
</table>

**Reading Data**

This section describes reading data from your serial port device in three parts:

- “The Input Buffer and Data Flow” on page 16-36 describes the flow of data from the device to MATLAB software.
- “Reading Text Data” on page 16-38 describes how to read from the device, and format the data as text.
• “Reading Binary Data” on page 16-40 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**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetl</td>
<td>Read one line of text from the device and discard the terminator</td>
</tr>
<tr>
<td>fgets</td>
<td>Read one line of text from the device and include the terminator</td>
</tr>
<tr>
<td>fread</td>
<td>Read binary data from the device</td>
</tr>
<tr>
<td>fscanf</td>
<td>Read data from the device and format as text</td>
</tr>
<tr>
<td>readsasync</td>
<td>Read data asynchronously from the device</td>
</tr>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations</td>
</tr>
</tbody>
</table>

The following table shows the properties associated with reading data.

**Properties Associated with Reading Data**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesAvailable</td>
<td>Number of bytes available in the input buffer</td>
</tr>
<tr>
<td>InputBufferSize</td>
<td>Size of the input buffer in bytes</td>
</tr>
<tr>
<td>ReadAsyncMode</td>
<td>Specify whether an asynchronous read operation is continuous or manual</td>
</tr>
<tr>
<td>Timeout</td>
<td>Waiting time to complete a read or write operation</td>
</tr>
<tr>
<td>TransferStatus</td>
<td>Indicate if an asynchronous read or write operation is in progress</td>
</tr>
<tr>
<td>ValuesReceived</td>
<td>Total number of values read from the device</td>
</tr>
</tbody>
</table>

**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:

```matlab
s = serial('COM1');
get(s,{'InputBufferSize','BytesAvailable'})
an = ...
    [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.

```matlab
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.

```matlab
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.

```matlab
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.

```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`.

```matlab
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.

```matlab
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.

```matlab
s.BytesAvailable
ans =
   69
```

To return the data to MATLAB, use any of the synchronous read functions. However, if you use fgets, fprintf, 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.

```matlab
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 16-38. 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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
```

Transfer data from the input buffer to MATLAB using `fscanf`.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
C = textscan(a, '%d%d%d%s%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.

```matlab
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.

```matlab
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.

```matlab
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 16-46
“Example — Introduction to Events and Callbacks” on page 16-46
“Event Types and Callback Properties” on page 16-47
“Respond To Event Information” on page 16-49
“Create and Execute Callback Functions” on page 16-51
“Enable Callback Functions After They Error” on page 16-52
“Example — Using Events and Callbacks” on page 16-52

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.

Note: All examples in this section on based on a Windows 32-bit platform. For information about other platforms refer to “Overview of a Serial Port Object” on page 16-22.

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.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Associated Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break interrupt</td>
<td>BreakInterruptFcn</td>
</tr>
<tr>
<td>Bytes available</td>
<td>BytesAvailableFcn</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnCount</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnMode</td>
</tr>
<tr>
<td>Error</td>
<td>ErrorFcn</td>
</tr>
<tr>
<td>Output empty</td>
<td>OutputEmptyFcn</td>
</tr>
<tr>
<td>Pin status</td>
<td>PinStatusFcn</td>
</tr>
<tr>
<td>Timer</td>
<td>TimerFcn</td>
</tr>
<tr>
<td></td>
<td>TimerPeriod</td>
</tr>
</tbody>
</table>
**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 16-6.

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 16-51, 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 16-60.

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**

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Field</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break interrupt</td>
<td>Type</td>
<td>BreakInterrupt</td>
</tr>
<tr>
<td>Break Interrupt</td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Bytes available</td>
<td>Type</td>
<td>BytesAvailable</td>
</tr>
<tr>
<td>Bytes Available</td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Error</td>
<td>Type</td>
<td>Error</td>
</tr>
<tr>
<td>Error</td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Error</td>
<td>Data.Message</td>
<td>An error string</td>
</tr>
<tr>
<td>Output empty</td>
<td>Type</td>
<td>OutputEmpty</td>
</tr>
<tr>
<td>Output Empty</td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Pin status</td>
<td>Type</td>
<td>PinStatus</td>
</tr>
<tr>
<td>Pin Status</td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Pin Status</td>
<td>Data.Pin</td>
<td>CarrierDetect, ClearToSend, DataSetReady, or RingIndicator</td>
</tr>
<tr>
<td>Pin Status</td>
<td>Data.PinValue</td>
<td>on or off</td>
</tr>
<tr>
<td>Timer</td>
<td>Type</td>
<td>Timer</td>
</tr>
<tr>
<td>Timer</td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
</tbody>
</table>

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 16-6.

**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:

```matlab
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @mycallback;
```

Alternatively, you can specify the callback function as a cell array.

```matlab
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:

```matlab
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:

```matlab
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.

   ```matlab
   s = serial('COM1');
   ```
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.

```matlab
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
s.OutputEmptyFcn = @instrcallback;
```

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.

```matlab
fopen(s)
```

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.

```matlab
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.

```matlab
out = fscanf(s)
out =
9600;0;0;NONE;LF
```

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.

```matlab
fclose(s)
delete(s)
clear s
```
Control Pins

In this section...

“Properties of Serial Port Control Pins” on page 16-54
“Signaling the Presence of Connected Devices” on page 16-54
“Controlling the Flow of Data: Handshaking” on page 16-57

Properties of Serial Port Control Pins

As described in “Serial Port Signals and Pin Assignments” on page 16-6, 9-pin serial ports include six control pins. The following table shows properties associated with the serial port control pins.

Control Pin Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataTerminalReady</td>
<td>State of the DTR pin</td>
</tr>
<tr>
<td>FlowControl</td>
<td>Data flow control method to use</td>
</tr>
<tr>
<td>PinStatus</td>
<td>State of the CD, CTS, DSR, and RI pins</td>
</tr>
<tr>
<td>RequestToSend</td>
<td>State of the RTS pin</td>
</tr>
</tbody>
</table>

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.

Note: All examples in this section are based on a Windows 32-bit platform. For more information about other supported platforms, refer to “Overview of a Serial Port Object” on page 16-22.
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.

\[
s1 = \text{serial('COM1')};
\]

\[
s2 = \text{serial('COM2')};
\]

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.

\[
s1.PinStatus
\]

\[
ans =
\]

\[
\begin{align*}
\text{CarrierDetect:} & \quad \text{'off'} \\
\text{ClearToSend:} & \quad \text{'on'} \\
\text{DataSetReady:} & \quad \text{'on'} \\
\text{RingIndicator:} & \quad \text{'off'}
\end{align*}
\]

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.

\[
s1.BaudRate = 2400;
\]

\[
s1.Terminator = \text{'CR'};
\]

\[
s2.BaudRate = 2400;
\]

\[
s2.Terminator = \text{'CR'};
\]
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
close([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
- Software handshaking

**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 16-8.

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**

<table>
<thead>
<tr>
<th>Character</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xon</td>
<td>17</td>
<td>Resume data transmission</td>
</tr>
<tr>
<td>Xoff</td>
<td>19</td>
<td>Pause data transmission</td>
</tr>
</tbody>
</table>

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 16-43. To do this, you must configure the oscilloscope and serial port object for software flow control.

```matlab
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 16-60
“Recording Properties” on page 16-60
“Example: Introduction to Recording Information” on page 16-61
“Creating Multiple Record Files” on page 16-61
“Specifying a Filename” on page 16-61
“The Record File Format” on page 16-62
“Example: Recording Information to Disk” on page 16-63

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.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordDetail</td>
<td>Amount of information saved to a record file</td>
</tr>
<tr>
<td>RecordMode</td>
<td>Specify whether data and event information is saved to one record file or to multiple record files</td>
</tr>
<tr>
<td>RecordName</td>
<td>Name of the record file</td>
</tr>
<tr>
<td>RecordStatus</td>
<td>Indicate if data and event information are saved to a record file</td>
</tr>
</tbody>
</table>
Note: All examples in this section are based on a Windows 32-bit platform. For more information about other supported platforms, refer to “Overview of a Serial Port Object” on page 16-22.

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.

```matlab
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.

close(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 RecordName. 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
Serial Port I/O

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

\[
value = (-1)^{\text{sign}} (2^{\text{exp}-127})(1.\text{significant})
\]

Double-precision floating-point values consist of 64 bits. The value is given by

\[
value = (-1)^{\text{sign}} (2^{\text{exp}-1023})(1.\text{significant})
\]
The floating-point format component, and the associated single-precision and double-precision bits are shown in the following table.

<table>
<thead>
<tr>
<th>Component</th>
<th>Single-Precision Bits</th>
<th>Double-Precision Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>exp</td>
<td>2–9</td>
<td>2–12</td>
</tr>
<tr>
<td>significand</td>
<td>10–32</td>
<td>13–64</td>
</tr>
</tbody>
</table>

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 16-41.

   ```
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 38 30 39 45 30 0a
10 Recording off.
Save and Load

In this section...

“Using save and load” on page 16-66
“Using Serial Port Objects on Different Platforms” on page 16-66

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.

```matlab
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`:

```matlab
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.

```matlab
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

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Disconnect a Serial Port Object” on page 16-68</td>
</tr>
<tr>
<td>“Clean Up the MATLAB Environment” on page 16-68</td>
</tr>
</tbody>
</table>

**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.

```matlab
close(s)
```

Examine the `Status` property to verify that the serial port object and the device are disconnected.

```matlab
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.

```matlab
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.

```matlab
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...

“The Property Reference Page Format” on page 16-70
“Serial Port Object Properties” on page 16-70

The Property Reference Page Format

Each serial port property description contains some or all of this information:

• The property name
• A description of the property
• The property characteristics, including:
  • Read only — The condition under which the property is read only
    A property can be read-only always, never, while the serial port object is open, or while the serial port object is recording. You can configure a property value using the \texttt{set} function or dot notation. You can return the current property value using the \texttt{get} function or dot notation.
  • Data type — the property data type
    This is the data type you use when specifying a property value.
  • Valid property values including the default value
    When property values are given by a predefined list, the default value is usually indicated by 
  • An example using the property
  • Related properties and functions

Serial Port Object Properties

The serial port object properties are briefly described below, and organized into categories based on how they are used. Following this section the properties are listed alphabetically and described in detail.
## Communications Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaudRate</td>
<td>Rate at which bits are transmitted</td>
</tr>
<tr>
<td>DataBits</td>
<td>Number of data bits to transmit</td>
</tr>
<tr>
<td>Parity</td>
<td>Type of parity checking</td>
</tr>
<tr>
<td>StopBits</td>
<td>Number of bits used to indicate the end of a byte</td>
</tr>
<tr>
<td>Terminator</td>
<td>Terminator character</td>
</tr>
</tbody>
</table>

## Write Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesToOutput</td>
<td>Number of bytes currently in the output buffer</td>
</tr>
<tr>
<td>OutputBufferSize</td>
<td>Size of the output buffer in bytes</td>
</tr>
<tr>
<td>Timeout</td>
<td>Waiting time to complete a read or write operation</td>
</tr>
<tr>
<td>TransferStatus</td>
<td>Indicate if an asynchronous read or write operation is in progress</td>
</tr>
<tr>
<td>ValuesSent</td>
<td>Total number of values written to the device</td>
</tr>
</tbody>
</table>

## Read Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesAvailable</td>
<td>Number of bytes available in the input buffer</td>
</tr>
<tr>
<td>InputBufferSize</td>
<td>Size of the input buffer in bytes</td>
</tr>
<tr>
<td>ReadAsyncMode</td>
<td>Specify whether an asynchronous read operation is continuous or manual</td>
</tr>
<tr>
<td>Timeout</td>
<td>Waiting time to complete a read or write operation</td>
</tr>
<tr>
<td>TransferStatus</td>
<td>Indicate if an asynchronous read or write operation is in progress</td>
</tr>
<tr>
<td>ValuesReceived</td>
<td>Total number of values read from the device</td>
</tr>
</tbody>
</table>

## Callback Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreakInterruptFcn</td>
<td>Callback function to execute when a break-interrupt event occurs</td>
</tr>
<tr>
<td>BytesAvailableFcn</td>
<td>Callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read</td>
</tr>
<tr>
<td><strong>Callback Properties</strong></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BytesAvailableFcnCount</td>
<td>Number of bytes that must be available in the input buffer to generate a bytes-available event</td>
</tr>
<tr>
<td>BytesAvailableFcnMode</td>
<td>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</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Callback function to execute when an error event occurs</td>
</tr>
<tr>
<td>OutputEmptyFcn</td>
<td>Callback function to execute when the output buffer is empty</td>
</tr>
<tr>
<td>PinStatusFcn</td>
<td>Callback function to execute when the CD, CTS, DSR, or RI pins change state</td>
</tr>
<tr>
<td>TimerFcn</td>
<td>Callback function to execute when a predefined period of time passes</td>
</tr>
<tr>
<td>TimerPeriod</td>
<td>Period of time between timer events</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Control Pin Properties</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DataTerminalReady</td>
<td>State of the DTR pin</td>
</tr>
<tr>
<td>FlowControl</td>
<td>Data flow control method to use</td>
</tr>
<tr>
<td>PinStatus</td>
<td>State of the CD, CTS, DSR, and RI pins</td>
</tr>
<tr>
<td>RequestToSend</td>
<td>State of the RTS pin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Recording Properties</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordDetail</td>
<td>Amount of information saved to a record file</td>
</tr>
<tr>
<td>RecordMode</td>
<td>Specify whether data and event information are saved to one record file or to multiple record files</td>
</tr>
<tr>
<td>RecordName</td>
<td>Name of the record file</td>
</tr>
<tr>
<td>RecordStatus</td>
<td>Indicate if data and event information are saved to a record file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>General Purpose Properties</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteOrder</td>
<td>Order in which the device stores bytes</td>
</tr>
<tr>
<td>Name</td>
<td>Descriptive name for the serial port object</td>
</tr>
<tr>
<td>General Purpose Properties</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Port</td>
<td>Platform-specific serial port name</td>
</tr>
<tr>
<td>Status</td>
<td>Indicate if the serial port object is connected to the device</td>
</tr>
<tr>
<td>Tag</td>
<td>Label to associate with a serial port object</td>
</tr>
<tr>
<td>Type</td>
<td>Object type</td>
</tr>
<tr>
<td>UserData</td>
<td>Data you want to associate with a serial port object</td>
</tr>
</tbody>
</table>
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 16-13.

**Characteristics**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

**Values**

The default value is 9600.
See Also

Properties

DataBits, Parity, StopBits
**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 16-51.

**Characteristics**

<table>
<thead>
<tr>
<th></th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

**Values**

The default value is an empty string.
See Also

Functions
record

Properties
RecordStatus
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>littleEndian</th>
<th>The byte order of the device is little-endian.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigEndian</td>
<td>The byte order of the device is big-endian.</td>
</tr>
</tbody>
</table>

See Also

Properties

Status
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 16-38.

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

<table>
<thead>
<tr>
<th>Characteristics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

Values

The default value is 0.

See Also

Functions

fopen
Properties

InputBufferSize, TransferStatus, ValuesReceived
**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

To learn how to create a callback function, see “Create and Execute Callback Functions” on page 16-51.

**Characteristics**

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

**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.

```matlab
s = serial('COM1');
```

Configure `s` to execute the callback function `instrcallback` when 40 bytes are available in the input buffer.

```matlab
s.BytesAvailableFcnCount = 40;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @instrcallback;
```

Connect `s` to the oscilloscope.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
s.BytesAvailable
ans =
   102
```

See Also

Functions

record

Properties

BytesAvailableFcnCount, BytesAvailableFcnMode, RecordStatus, Terminator, TransferStatus
**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**

<table>
<thead>
<tr>
<th>Read only</th>
<th>While open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

**Values**

The default value is 48.

**See Also**

**Functions**

`fclose`
Properties

BytesAvailableFcn, BytesAvailableFcnMode, Status
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>While open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>{terminator}</th>
<th>A bytes-available event is generated when the terminator is read.</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>A bytes-available event is generated when the specified number of bytes are available.</td>
</tr>
</tbody>
</table>
See Also

Functions
fclose

Properties
BytesAvailableFcn, BytesAvailableFcnCount, Status, Terminator
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 16-32.

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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

Values

The default value is 0.
See Also

Functions
fopen

Properties
OutputBufferSize, TransferStatus, ValuesSent
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 16-13.

Characteristics

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

Values

DataBits can be 5, 6, 7, or 8. The default value is 8.
See Also

Properties

Parity, StopBits
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 16-57.

Characteristics

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>{on}</th>
<th>The DTR pin is asserted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>The DTR pin is unasserted.</td>
</tr>
</tbody>
</table>

See Also

Properties

FlowControl, PinStatus
**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 16-51.

**Characteristics**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

**Values**

The default value is an empty string.
See Also

Functions
record

Properties
RecordStatus, Timeout
**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 16-57.

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**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>{none}</th>
<th>No flow control is used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardware</td>
<td>Hardware flow control is used.</td>
</tr>
</tbody>
</table>
FlowControl

| software | Software flow control is used. |

See Also

Properties

PinStatus, RequestToSend
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 fgets, fopen, 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

<table>
<thead>
<tr>
<th>Read only</th>
<th>While open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

Values

The default value is 512.

See Also

Functions

fclose, fgets, fopen, fread, fscanf
Properties

Status
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

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.

```matlab
s = serial('COM1');
```

s is automatically assigned a descriptive name.

```matlab
s.Name
ans =
```
Serial-COM1

See Also

Functions

serial
**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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Any serial port object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>{on}</th>
<th>Object is visible to <code>instrfind</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>off</code></td>
<td>Object is not visible from the command line (except by <code>instrfindall</code>).</td>
</tr>
</tbody>
</table>

**Examples**

The following statement creates a serial port object with its `ObjectVisibility` property set to `off`:

```matlab
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**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>While open</td>
</tr>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

**Values**

The default value is 512.

**See Also**

**Functions**

`fprintf`, `fwrite`

**Properties**

`Status`
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 16-51.

Characteristics

| Read only | Never |
| Data type | Callback function |

Values

The default value is an empty string.
See Also

Functions
record

Properties
RecordStatus
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 16-12.

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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{none}</td>
<td>No parity checking</td>
</tr>
<tr>
<td>odd</td>
<td>Odd parity checking</td>
</tr>
<tr>
<td>even</td>
<td>Even parity checking</td>
</tr>
<tr>
<td>mark</td>
<td>Mark parity checking</td>
</tr>
<tr>
<td>space</td>
<td>Space parity checking</td>
</tr>
</tbody>
</table>

**See Also**

**Properties**

DataBits, StopBits
**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 16-6.

*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 16-55.

**Characteristics**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Structure</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>off</th>
<th>The associated pin is unasserted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>The associated pin is asserted.</td>
</tr>
</tbody>
</table>
The default value is device dependent.

See Also

Properties

DataTerminalReady, PinStatusFcn, RequestToSend
**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 16-51.

**Characteristics**

<table>
<thead>
<tr>
<th></th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

**Values**

The default value is an empty string.
See Also

Functions
record

Properties
PinStatus, RecordStatus
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>While open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

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.

```matlab
s = serial('COM1');
```

The value of the Port property is COM1.

```matlab
s.Port
ans =
```
COM1

See Also

Functions
fclose, serial

Properties
Name, Status
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

| {continuous}                      | Continuously query the device to determine if data is available to be read. |
| manual | Manually read data from the device using the readasync function. |

See Also

Functions

fgets, fread, fscanf, readasync

Properties

BytesAvailable, InputBufferSize
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 16-63.

Characteristics

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>compact</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>verbose</td>
<td>The data written to the device, and the data read from the device are also saved to the record file.</td>
</tr>
</tbody>
</table>

See Also

Functions

record
Properties

RecordMode, RecordName, RecordStatus
**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 16-61.

**Characteristics**

<table>
<thead>
<tr>
<th>Read only</th>
<th>While recording</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>{overwrite}</th>
<th>The record file is overwritten.</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Data is appended to an existing record file.</td>
</tr>
<tr>
<td>index</td>
<td>A different record file is created, each with an indexed filename.</td>
</tr>
</tbody>
</table>

**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

See Also

Functions

record

Properties

RecordDetail, RecordName, RecordStatus
**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.

```matlab
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 16-61.

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**

<table>
<thead>
<tr>
<th>Read only</th>
<th>While recording</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

The default record filename is `record.txt`. 
See Also

Functions
record

Properties
RecordDetail, RecordMode, RecordStatus
**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 16-60.

**Characteristics**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>{off}</th>
<th>Data and event information are not written to a record file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Data and event information are written to a record file.</td>
</tr>
</tbody>
</table>

**See Also**

**Functions**

`record`
Properties

RecordDetail, RecordMode, RecordName
**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 16-57.

**Characteristics**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>{on}</th>
<th>The RTS pin is asserted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>The RTS pin is unasserted.</td>
</tr>
</tbody>
</table>
See Also

Properties

FlowControl, PinStatus
**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**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>{closed}</th>
<th>The serial port object is not connected to the device.</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>The serial port object is connected to the device.</td>
</tr>
</tbody>
</table>

**See Also**

**Functions**

fclose, fopen
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 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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>{1}</th>
<th>One stop bit is transmitted to indicate the end of a byte.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>The stop bit is transferred for 150% of the normal time used to transfer one bit.</td>
</tr>
<tr>
<td>2</td>
<td>Two stop bits are transmitted to indicate the end of a byte.</td>
</tr>
</tbody>
</table>
See Also

Properties

DataBits, Parity
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

The default value is an empty string.

Example

Suppose you create a serial port object associated with the serial port COM1.

```matlab
s = serial('COM1');
fopen(s)
```

You can assign s a unique label using Tag.

```matlab
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.

```matlab
s1 = instrfind('Tag','MySerialObj');
```

See Also

Functions

`instrfind`
**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**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**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.
See Also

Functions
fgetl, fgets, fprintf, fscanf

Properties
BytesAvailableFcnMode
**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**

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

**Values**

The default value is 10 seconds.

**See Also**

**Properties**

ErrorFcn
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 16-51.

Characteristics

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

Values

The default value is an empty string.
See Also

Functions

fopen, record

Properties

RecordStatus, TimerPeriod
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

Values

The default value is 1 second. The minimum value is 0.01 second.

See Also

Functions

fopen

Properties

TimerFcn
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 16-28.

Characteristics

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<p>| {idle}          | No asynchronous operations are in progress. |
| read            | An asynchronous read operation is in progress. |</p>
<table>
<thead>
<tr>
<th>write</th>
<th>An asynchronous write operation is in progress.</th>
</tr>
</thead>
<tbody>
<tr>
<td>read&amp;write</td>
<td>Asynchronous read and write operations are in progress.</td>
</tr>
</tbody>
</table>

**See Also**

**Functions**

fprintf, fwrite, readasync

**Properties**

ReadAsyncMode
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

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

16-140
s     1x1     644  serial object

Grand total is 18 elements using 644 bytes

See Also

Functions

serial
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

<table>
<thead>
<tr>
<th>Read only</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Any type</td>
</tr>
</tbody>
</table>

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 16-10.

**Characteristics**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

**Values**

The default value is 0.

**Example**

Suppose you create a serial port object associated with the serial port COM1.

```plaintext
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.

```matlab
fprintf(s,'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
ans =
17
```

See Also

**Functions**

`fopen`

**Properties**

`BytesAvailable`
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 16-10.

Characteristics

<table>
<thead>
<tr>
<th>Read only</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

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.

```matlab
fprintf(s,'*IDN?')
s.ValuesSent
ans =
    6
```

**See Also**

**Functions**

fopen

**Properties**

BytesToOutput
Hardware Support

• “Support Package Installation” on page 17-2
• “Support Package Installer Help” on page 17-5
• “MATLAB Supported Hardware” on page 17-19
Support Package Installation

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 Support Package Installer, a wizard that guides you through the installation process. You can also use Support Package Installer to update installed support package software or update the firmware on third-party hardware.

To start Support Package Installer:

Install Downloaded Support Package on Multiple Computers

You can download a support package to one computer, and then install it on other computers. This allows you to:

• Install support packages on computers that are not connected to the Internet.
• Reduce the number of downloads you need to perform.

The computer you use for downloading must have the same base product license and platform as the computers upon which you are installing the support package. For example, suppose you want to install a Simulink support package on a group of computers that are running 64-bit Windows. To do so, you must first download the support package using a computer that has a Simulink license and is running 64-bit Windows.

To install a downloaded support package on multiple computers:

1 Download the support package, and place the folder in a shared network location.
2 Run Support Package Installer on the other computers.
3 On the Install or update support package screen, select the Folder option.
Follow the instructions provided by Support Package Installer to complete the installation process.

**Note:** Some support packages require you to install third-party software. If so, ensure that the third-party software is available for installation on each computer.

**Troubleshoot Timed Out Connections**

While you are installing a support package, Support Package Installer downloads software from the Internet. If downloading the software takes too long, Support Package Installer displays:

“Unable to download http:<url>. Connection timed out. Verify your network connection and proxy settings. If they are properly configured, retry the support package installation after 15 minutes. If the problem persists, contact MathWorks Technical Support.”

In some cases, antivirus software on the host computer causes this error by delaying or blocking the software download process. Consider pausing the antivirus software, or changing the antivirus software settings. Then, try installing the support package again.

**See Also**
supportPackageInstaller | targetupdater

**External Websites**

- Hardware Support Catalog
Support Package Installer Help

In this section...

“About Support Package Installer” on page 17-5
“Select an Action” on page 17-6
“Select Support Package to Install or Select Support Package to Download” on page 17-7
“Log In to MathWorks Account” on page 17-10
“The MathWorks, Inc. Software License Agreement” on page 17-12
“Third-Party Software Licenses” on page 17-13
“Confirm Installation, Confirm Download, Confirm Uninstall” on page 17-14
“Install or Update Complete, Download Complete, Uninstall Complete” on page 17-15
“Set Up Support Package” on page 17-16
“Support Package Setup Complete” on page 17-18
“Special Instructions” on page 17-18

About Support Package Installer

Use Support Package Installer to add support for third-party products to your MathWorks software. When you complete the installation process, you can use the MathWorks software with those third-party products.

To start Support Package Installer, enter supportPackageInstaller in a MATLAB Command Window.

The following topics provide help for each screen in the Support Package Installer.
Select an Action

You can use Select an action to choose the action Support Package Installer performs:

- **Install from Internet**: Download a support package from the Internet and install it on the host computer.
- **Download from Internet**: Download a support package from the Internet without installing it.
- **Install from folder**: Install a previously downloaded support package.
- **Uninstall**: Uninstall a specific support package.
For instructions on using **Download from Internet** and **Install from Folder**, see “Install Downloaded Support Package on Multiple Computers” on page 17-3.

**Select Support Package to Install or Select Support Package to Download**

Support Package Installer installs those support package or packages that are currently visible in the right pane. In other words, it only installs support for one type of hardware at a time.

**Use Support for** to choose a specific group of support packages. Support Package Installer can only install or act upon one particular group of support packages at a time.
You can use **Show** to filter the support packages Support Package Installer displays:

- **All** displays support packages for all base products and host computer platforms.
- **Installable** displays support packages that you can install with the product licenses and platform on your host computer.
- **Installed** displays support packages that are installed on your host computer.
- **Updatable** displays installed support packages for which updates are available.

To install support packages in multiple groups, rerun Support Package Installer for each group.

Use **Support packages** to choose and review the status of specific support packages. Support Package Installer organizes the following support package information in columns:

- **Action** displays the action Support Package Installer can perform upon a support package. Select or clear the check box in to enable or disable Support Package Installer from performing that action: **Install, Reinstall, Update, None**.
- **Installed Version** displays the version of the installed support package. This entry is blank if the support package is not installed. If Latest Version and Installed Version are different, you can update the support package.
- **Latest Version** displays the version of the support package that is available for download. If Latest Version and Installed Version are different, you can update the support package. If your computer does not have access to the Internet, this value is empty.
- **Description** links to a web page where you can look or search for more information about the support package.
- **Required Base Product** displays the base product that is required in order to install the support package. If the base product is not licensed and installed, the support package is not downloadable or installable.
- **Supported Host Platforms** displays the host computer platforms required to install the support package. If the host computer is not one of the supported platforms, you cannot download or install the support package. **Win32** and **Win64** represent the 32- and 64-bit versions of Microsoft Windows. **Linux64** represents the 64-bit version of Linux. **Mac64** represents the 64-bit version of the Apple Mac platform.

The **Installation folder** parameter displays the path where Support Package Installer installs the support package. You must have write privileges for this folder. You can
enter a new path directly, or use the **Browse** button to select one. The installation process creates a **downloads** folder in this location.

If you choose the **Download from Internet** option, Support Package Installer displays the **Download folder** parameter instead of the **Installation folder** parameter. **Download folder** displays the path where Support Package Installer creates support package folders. You must have write privileges for this path. You can enter a new path directly, or use the **Browse** button to select one. By default, the **Download folder** is named **downloads**. After installing support packages, the **Download folder** contains subfolders for each support package.

The **Find more supported hardware** link opens a browser and displays the Hardware Support topic on the MathWorks website.

**Note:** Opening a support package installation file opens the **Select a Support Package to Install or Download** screen with a support package preselected. These files are available from MATLAB Central File Exchange, and have the file name extension: ***.mlpkginstall**
Log In to MathWorks Account

Click **Log In**. In the **MathWorks Account Log In** dialog box, enter your MathWorks.com account email address and password. For more information, see [http://www.mathworks.com/support/solutions/en/data/1-172ME/index.html?solution=1-172ME](http://www.mathworks.com/support/solutions/en/data/1-172ME/index.html?solution=1-172ME).

If you do not have an account, click here to create one.
The MathWorks, Inc. Software License Agreement

Review the text of the license agreement. If you agree, select the I accept check box, and then click Next.
Third-Party Software Licenses

Support packages that download third-party software as part of the installation process display this screen. Review the information and any license agreements.

To proceed, click Install. Otherwise, click Back or Cancel.
Confirm Installation, Confirm Download, Confirm Uninstall

This screen confirms the name of the support package you have chosen. To continue, click Install, Download, or Uninstall.

Support Package Installer displays a progress bar while it performs the specified action.

**Note:** If Support Package Installer instructs you to delete files, close the MATLAB software before removing the files. Then, restart MATLAB software and run Support Package Installer to complete the process.
Install or Update Complete, Download Complete, Uninstall Complete

Support Package Installer has finished installing or uninstalling the support package.

Some support packages require additional actions before you can use the support package:

• Replace or update the firmware on the target hardware.
• Install and validate required third-party software.

To perform these actions, click Continue. Otherwise, click Finish.
This screen displays the path of the support package folder within the **Download folder**. If you use the **Install from Folder** option in the future, you will specify the location of this support package folder.

If there are no additional actions, and the support package contains featured examples, Support Package Installer gives you the option to open the featured examples. These examples show you how to use the capabilities provided by the support package.

**Set Up Support Package**
This screen is only shown for support packages that provide additional post-installation steps. Use the **Support package for** parameter to identify the third-party hardware or software to set up. Then, click **Next**.

The subsequent screens guide you through the process of:

- Replacing or updating the firmware on the target hardware.
- Installing and validating required third-party software.

**Note:** To return to this screen without repeating preceding screens, enter `targetupdater` in a MATLAB Command Window.
Support Package Setup Complete

You have completed the set up process.

If you leave the **Show support package examples** check box selected and click **Finish**, Support Package Installer displays support package examples in a new window.

**Special Instructions**

This screen is unique to the support package you are installing. Carefully read and follow the instructions provided by the screen.
MATLAB Supported Hardware

For a complete list of support packages, see the Hardware Support Catalog.

In addition to these packages, MATLAB includes built-in support for:

• “Audio Recording and Playback”
• “Serial Port Devices”