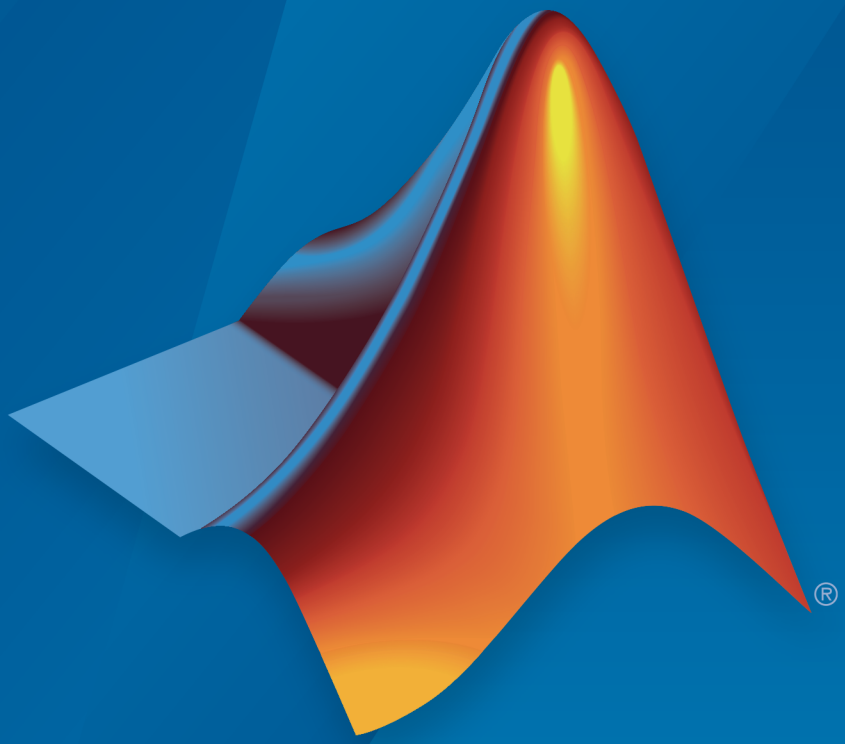
MATLAB® Compiler SDK™

COM User's Guide



MATLAB®

R2016a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

MATLAB® Compiler SDK™ COM User's Guide

Revision History

March 2015	Online only	New for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)

Contents

1 Create and Install COM Components

Package a Deployable COM Component 1-2
Add-In and COM Component Registration 1-2

Install COM Components 1-4

2 Programming with COM Components

General Techniques 2-2

Register and Reference the Utility Library 2-3

Call the Methods of a Class Instance 2-4
Standard Mapping Technique 2-4
Variant 2-5
Pass Input and Output Parameters 2-5

Call COM Objects in Visual C++ Programs 2-7

Pass Arguments 2-8
Overview 2-8
Create and Use a varargin Array in Microsoft Visual Basic Programs 2-8
Create and Use varargout in Microsoft Visual Basic Programs 2-9
Pass an Empty varargin From Microsoft Visual Basic Code . . 2-9

Control Array Formatting and Data Conversion 2-11
Overview 2-11

Array Formatting Flags 2-11
Using Array Formatting Flags 2-12
Using Data Conversion Flags 2-14
Special Flags for Some Microsoft Visual Basic Types 2-16

Use MATLAB Global Variables in Visual Basic 2-17

Block Execution of Applications that Create Figures 2-20
MCRWaitForFigures 2-20
Use MCRWaitForFigures to Block Execution 2-20

MATLAB Runtime Options 2-23
What MATLAB Runtime Options are Supported for COM? . 2-23
How Do I Specify MATLAB Runtime Options? 2-23

Share MATLAB Runtime Instances 2-24
What Is a Singleton MATLAB Runtime? 2-24
Advantages and Disadvantages of Using a Singleton 2-24

Obtain Registry Information 2-25

Handle Errors During a Method Call 2-27

Integrate Magic Square into a COM Application 2-28
Overview 2-28
Creating the MATLAB File 2-28
Using the Library Compiler App to Create and Build the Project 2-28
Creating the Microsoft Visual Basic Project 2-29
Creating the User Interface 2-29
Creating the Executable in Microsoft Visual Basic 2-31
Testing the Application 2-31

3 How the MATLAB Compiler SDK Product Creates COM Components

Overview of Internal Processes 3-2
Code Generation 3-2
Create Interface Definitions 3-2

C++ Compilation 3-2
Linking and Resource Binding 3-3
Registration of the DLL 3-3

Component Registration 3-4
Self-Registering Components 3-4
Globally Unique Identifier 3-5
Versioning 3-6

Data Conversion 3-8
Conversion Rules 3-8
Array Formatting Flags 3-17
Data Conversion Flags 3-18

Calling Conventions 3-20
Producing a COM Class 3-20
IDL Mapping 3-21
Microsoft Visual Basic Mapping 3-22

4 Distribute Integrated COM Applications

Package COM Applications 4-2

About the MATLAB Runtime 4-3
How is the MATLAB Runtime Different from MATLAB? 4-3
Performance Considerations and the MATLAB Runtime 4-4

Download the MATLAB Runtime Installer 4-5

Install the MATLAB Runtime 4-6
Install the MATLAB Runtime Interactively 4-6
Install the MATLAB Runtime Non-Interactively 4-7

MATLAB and MATLAB Runtime on Same Machine 4-10
Modifying the Path 4-10

Multiple MATLAB Runtime Versions on Single Machine . . 4-11

Uninstall MATLAB Runtime **4-12**
Windows **4-12**
Linux **4-12**
Mac **4-12**

5 Utility Library for Microsoft COM Components

Reference Utility Classes **5-2**

Class MWUtil **5-3**
Sub MWInitApplication(pApp As Object) **5-3**
Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList]) **5-5**
Function IsMCRJVMEnabled() As Boolean **5-6**
Function IsMCRInitialized() As Boolean **5-6**
Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31]) **5-7**
Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31]) **5-8**
Sub MWDate2VariantDate(pVar) **5-10**

Class MWFlags **5-12**
Property ArrayFormatFlags As MWArrayFormatFlags **5-12**
Property DataConversionFlags As MWDataConversionFlags **5-15**
Sub Clone(ppFlags As MWFlags) **5-17**

Class MWStruct **5-19**
Sub Initialize([varDims], [varFieldNames]) **5-19**
Property Item([i0], [i1], ..., [i31]) As MWField **5-20**
Property NumberOfFields As Long **5-23**
Property NumberOfDims As Long **5-23**
Property Dims As Variant **5-23**
Property FieldNames As Variant **5-23**
Sub Clone(ppStruct As MWStruct) **5-24**

Class MWField **5-26**
Property Name As String **5-26**
Property Value As Variant **5-26**
Property MWFlags As MWFlags **5-26**
Sub Clone(ppField As MWField) **5-26**

Class MWComplex **5-28**
Property Real As Variant **5-28**
Property Imag As Variant **5-28**
Property MWFlags As MWFlags **5-29**
Sub Clone(ppComplex As MWComplex) **5-29**

Class MWSparse **5-31**
Property NumRows As Long **5-31**
Property NumColumns As Long **5-31**
Property RowIndex As Variant **5-31**
Property ColumnIndex As Variant **5-32**
Property Array As Variant **5-32**
Property MWFlags As MWFlags **5-32**
Sub Clone(ppSparse As MWSparse) **5-32**

Class MWArg **5-35**
Property Value As Variant **5-35**
Property MWFlags As MWFlags **5-35**
Sub Clone(ppArg As MWArg) **5-35**

Enum mwArrayFormat **5-37**

Enum mwDataType **5-38**

Enum mwDateFormat **5-39**

6 Functions — Alphabetical List

Create and Install COM Components

- "Package a Deployable COM Component" on page 1-2
- "Install COM Components" on page 1-4

Package a Deployable COM Component

Add-In and COM Component Registration

Note: COM components are used in both MATLAB Compiler™ and MATLAB Compiler SDK, therefore some of the instructions relating to building and packaging COM components and add-ins can be shared between products.

When you create your COM component, it is registered in either `HKEY_LOCAL_MACHINE` or `HKEY_CURRENT_USER`, based on your log-in privileges.

If you find you need to change your run-time permissions due to security standards imposed by Microsoft® or your installation, you can do one of the following before deploying your COM component or add-in:

- Log on as `administrator` before running your COM component or add-in
- Run the following `mwregsvr` command prior to running your COM component or add-in, as follows:

  ```
  mwregsvr [/u] [/s] [/useronly] project_name.dll
  ```
  where:

  - `/u` allows any user to unregister a COM component or add-in for this server
  - `/s` runs this command silently, generating no messages. This is helpful for use in silent installations.
  - `/useronly` allows only the currently logged-in user to run the COM component or add-in on this server

Caution If your COM component is registered in the `USER` hive, it will not be visible to Windows Vista™ or Windows® 7 users running as `administrator` on systems with UAC (**User Access Control**) enabled.

If you register a component to the `USER` hive under Windows 7 or Windows Vista, your COM component may fail to load when running with elevated (`administrator`) privileges.

If this occurs, do the following to re-register the component to the `LOCAL MACHINE` hive:

1. Unregister the component with this command:

   ```
   mwregsvr /u /useronly my_dll.dll
   ```

2. Reregister the component to the `LOCAL MACHINE` hive with this command:

   ```
   mwregsvr my_dll.dll
   ```

Install COM Components

To install and deploy a COM object created with MATLAB Compiler SDK:

1. Install the MATLAB Runtime as described in “Install MATLAB Runtime”.
2. Build and package as described in “Compile COM Components with Library Compiler App” and “Package a Deployable COM Component” on page 1-2.
3. Copy the package to the target computer and run the package.
4. From a Windows command prompt on the target system, navigate to the folder where you saved the package. If you use the command `dir`, you should see the `.dll` created for your COM object. You will need to register the `.dll` manually using the command `regsvr32`, as follows:

   ```
   regsvr32 myCom_1_0.dll
   ```

Programming with COM Components

- "General Techniques" on page 2-2
- "Register and Reference the Utility Library" on page 2-3
- "Call the Methods of a Class Instance" on page 2-4
- "Call COM Objects in Visual C++ Programs" on page 2-7
- "Pass Arguments " on page 2-8
- "Control Array Formatting and Data Conversion" on page 2-11
- "Use MATLAB Global Variables in Visual Basic" on page 2-17
- "Block Execution of Applications that Create Figures" on page 2-20
- "MATLAB Runtime Options" on page 2-23
- "Share MATLAB Runtime Instances" on page 2-24
- "Obtain Registry Information" on page 2-25
- "Handle Errors During a Method Call" on page 2-27
- "Integrate Magic Square into a COM Application" on page 2-28

General Techniques

After you package and install a COM component, you can access the component in any program that supports COM, such as Microsoft Visual Basic®, Microsoft Visual C++®, or Visual C#.

Your code module must do the following:

- Load the components created by the compiler
  - "Register and Reference the Utility Library" on page 2-3
- Call methods of the component class
  - "Call the Methods of a Class Instance" on page 2-4
  - "Call COM Objects in Visual C++ Programs" on page 2-7
  - "Obtain Registry Information" on page 2-25
- Deal with data conversion and parameter passing
  - "Pass Arguments " on page 2-8
  - "Control Array Formatting and Data Conversion" on page 2-11
  - "Use MATLAB Global Variables in Visual Basic" on page 2-17
- Process errors
  - "Handle Errors During a Method Call" on page 2-27

Note: These topics provide general information on how to integrate COM components created with the compiler into your COM-compliant programs. The presentation focuses on the special programming techniques needed for components based on the MATLAB product and generated by the compiler. It assumes that you have a working knowledge of the programming language used in these programs.

For information about programming with COM objects in Microsoft Visual Studio®, see articles in the MSDN Library, such as Calling COM Components from .NET Clients.

Register and Reference the Utility Library

The `MWComUtil` library provided with MATLAB Compiler SDK is freely distributable. The `MWComUtil` library includes seven classes and three enumerated types. These utilities are required for array processing, and they provide type definitions used in data conversion.

The library is contained in the file `mwcomutil.dll`. It must be registered once on each machine that uses components created with the compiler.

If you are on a development machine that does not have an installation of MATLAB Compiler SDK, register the `MWComUtil` library at the system prompt with the command:

```
mwregsvr mwcomutil.dll
```

To use the types in the library, make sure that you reference the `MWComUtil` library in your current project.

Note: You must specify the full path of the component when calling `mwregsvr`, or make the call from the folder in which the component resides. `mwregsvr.exe` is supplied with the MATLAB Runtime.

Call the Methods of a Class Instance

In this section...
“Standard Mapping Technique” on page 2-4
“Variant” on page 2-5
“Pass Input and Output Parameters” on page 2-5

Standard Mapping Technique

After you create a class instance, you can call the class methods to access the encapsulated MATLAB functions. The MATLAB Compiler SDK product uses a standard technique to map the original MATLAB function syntax to the method's argument list. This standard mapping technique is as follows:

- `nargout`

  When a method has output arguments, the first argument is always `nargout`, which is of type `Long`. This input parameter passes the normal MATLAB `nargout` parameter to the encapsulated function and specifies how many outputs are requested. Methods that do not have output arguments do not pass a `nargout` argument.

- Output parameters

  Following `nargout` are the output parameters listed in the same order as they appear on the left side of the original MATLAB function.

- Input parameters

  Next come the input parameters listed in the same order as they appear on the right side of the original MATLAB function.

For example, the most generic MATLAB function is:

```
function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)
```

This function maps directly to the following Microsoft Visual Basic signature:

```
Sub foo(nargout As Long, _
        Y1 As Variant, _
        Y2 As Variant, _
```

```
.
.
varargout As Variant, _
X1 As Variant, _
X2 As Variant, _
.
.
varargin As Variant)
```

See “Calling Conventions” on page 3-20 for more details and examples of the standard mapping from MATLAB functions to COM class method calls.

Variant

All input and output arguments are typed as `Variant`, the default Visual Basic data type. The `Variant` type can hold any of the basic Visual Basic types, arrays of any type, and object references. See “Data Conversion” on page 3-8 for details about the conversion of any basic type to and from MATLAB data types.

In general, you can supply any Visual Basic type as an argument to a class method, with the exception of Visual Basic User Defined Types (UDTs).

When you pass a simple `Variant` type as an output parameter, the called method allocates the received data and frees the original contents of the `Variant`. In this case it is sufficient to dimension each output argument as a single `Variant`. When an object type (like an Excel® `Range`) is passed as an output parameter, the object reference is passed in both directions, and the object's `Value` property receives the data.

Pass Input and Output Parameters

The following examples show how to pass input and output parameters to COM component class methods in Visual Basic.

The first example is a function, `foo`, that takes two arguments and returns one output argument. The `foo` function dispatches a call to a class method that corresponds to a MATLAB function of the form `function y = foo(x1,x2)`.

```
Function foo(x1 As Variant, x2 As Variant) As Variant
   Dim aClass As Object
   Dim y As Variant
```

```
   On Error Goto Handle_Error
   Set aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,y,x1,x2)
   foo = y
   Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

The second example rewrites the `foo` function as a subroutine:

```
Sub foo(Xout As Variant, X1 As Variant, X2 As Variant)
   Dim aClass As Object

   On Error Goto Handle_Error
   Set aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,Xout,X1,X2)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

Call COM Objects in Visual C++ Programs

Note: You must choose a Microsoft compiler to compile and use any COM object.

Use the COM object you have created as follows:

1. Create a Visual C++ program in a file named `matlab_com_example.cpp` with the following code:

```
#include <iostream>
using namespace std;

#include "mycomponent\src\mycomponent_idl.h"
#include "mycomponent\src\mycomponent_idl_i.c"

int main() {
// Initialize argument variables
              VARIANT x, y, out1;
//Initialize the COM library
              HRESULT hr = CoInitialize(NULL);
//Create an instance of the COM object you created
              Imycomponentclass *pImycomponentclass;
              hr=CoCreateInstance
                (CLSID_mycomponentclass, NULL, CLSCTX_INPROC_SERVER,
                 IID_Imycomponentclass,(void **)&pImycomponentclass);
// Set the input arguments to the COM method
              x.vt=VT_R8;
              y.vt=VT_R8;
              x.dblVal=7.3;
              y.dblVal=1946.0;
// Access the method with arguments and receive the output out1
              hr=(pImycomponentclass -> adddoubles(1,&out1,x,y));
// Print the output
              cout << "The input values were " << x.dblVal << " and "
                   << y.dblVal << ".\n";
              cout << "The output of feeding the inputs into the adddoubles method is "
                   << out1.dblVal << ".\n";
// Uninitialize COM
              CoUninitialize();
              return 0;
}
```

2. In the MATLAB Command Window, compile the program as follows:

```
mbuild matlab_com_example.cpp
```

When you run the executable, the program displays two numbers and their sum, as returned by the COM object's `adddoubles`.

Pass Arguments

In this section...

"Overview" on page 2-8
"Create and Use a varargin Array in Microsoft Visual Basic Programs" on page 2-8
"Create and Use varargout in Microsoft Visual Basic Programs" on page 2-9
"Pass an Empty varargin From Microsoft Visual Basic Code" on page 2-9

Overview

When it encapsulates MATLAB functions, the MATLAB Compiler SDK product adds the MATLAB function arguments to the argument list of the class methods it creates. Thus, if a MATLAB function uses `varargin` and/or `varargout`, the compiler adds these arguments to the argument list of the class method. They are added at the end of the argument list for input and output arguments.

You can pass multiple arguments as a `varargin` array by creating a `Variant` array, assigning each element of the array to the respective input argument.

See "Producing a COM Class" on page 3-20 for more information about mapping of input and output arguments.

Create and Use a varargin Array in Microsoft Visual Basic Programs

The following example creates a `varargin` array to call a method encapsulating a MATLAB function of the form `y=foo(varargin)`.

The `MWUtil` class included in the `MWComUtil` utility library provides the `MWPack` helper function to create `varargin` parameters.

```
Function foo(x1 As Variant, x2 As Variant, x3 As Variant, _
             x4 As Variant, x5 As Variant) As Variant
   Dim aClass As Object
   Dim v(1 To 5) As Variant
   Dim y As Variant

   On Error Goto Handle_Error
   v(1) = x1
```

```
   v(2) = x2
   v(3) = x3
   v(4) = x4
   v(5) = x5
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,y,v)
   foo = y
   Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

Create and Use varargout in Microsoft Visual Basic Programs

The next example processes a `varargout` argument as three separate arguments. This function uses the `MWUnpack` function in the utility library.

The MATLAB function used is `varargout=foo(x1,x2)`.

```
Sub foo(Xout1 As Variant, Xout2 As Variant, Xout3 As Variant, _
        Xin1 As Variant, Xin2 As Variant)
   Dim aClass As Object
   Dim aUtil As Object
   Dim v As Variant

   On Error Goto Handle_Error
   aUtil = CreateObject("MWComUtil.MWUtil")
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(3,v,Xin1,Xin2)
   Call aUtil.MWUnpack(v,0,True,Xout1,Xout2,Xout3)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

Pass an Empty varargin From Microsoft Visual Basic Code

In MATLAB, `varargin` inputs to functions are optional, and may be present or omitted from the function call. However, from Microsoft Visual Basic, function signatures are more strict—if `varargin` is present among the MATLAB function inputs, the VBA call must include `varargin`, even if you want it to be empty. To pass in an empty `varargin`, pass the `Null` variant, which is converted to an empty MATLAB cell array when passed.

Passing an Empty varargin From VBA Code

The following example illustrates how to pass the null variant in order to pass an empty `varargin`:

```
Function foo(x1 As Variant, x2 As Variant, x3 As Variant, _
             x4 As Variant, x5 As Variant) As Variant
   Dim aClass As Object
   Dim v(1 To 5) As Variant
   Dim y As Variant

   On Error Goto Handle_Error
   v(1) = x1
   v(2) = x2
   v(3) = x3
   v(4) = x4
   v(5) = x5
   aClass = CreateObject("mycomponent.myclass.1_0")

   'Call aClass.foo(1,y,v)
   Call aClass.foo(1,y,Null)

   foo = y
   Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

Control Array Formatting and Data Conversion

In this section...
"Overview" on page 2-11
"Array Formatting Flags" on page 2-11
"Using Array Formatting Flags" on page 2-12
"Using Data Conversion Flags" on page 2-14
"Special Flags for Some Microsoft Visual Basic Types" on page 2-16

Overview

Generally, you should write your application code so that it matches the arguments (input and output) of the MATLAB functions that are encapsulated in the COM objects that you are using. The mapping of arguments from the MATLAB product to Microsoft Visual Basic is fully described in MATLAB to COM VARIANT Conversion Rules and COM VARIANT to MATLAB Conversion Rules.

In some cases it is not possible to match the two kinds of arguments exactly; for example, when existing MATLAB code is used in conjunction with a third-party product such as Microsoft Excel. For these and other cases, the compiler supports formatting and conversion flags that control how array data is formatted in both directions (input and output).

When it creates a component, the compiler includes a component property named `MWFlags`. The `MWFlags` property is readable and writable.

The `MWFlags` property consists of two sets of constants: *arrayformattingflags* and *dataconversionflags*. Array formatting flags affect the transformation of arrays, whereas data conversion flags deal with type conversions of individual array elements.

Array Formatting Flags

The following tables provide a quick overview of how to use array formatting flags to specify conversions for input and output arguments.

Name of Flag	Possible Values of Flag	Results of Conversion
`InputArrayFormat`	`mwArrayFormatMatrix` (default)	MATLAB matrix from general `Variant` data.

Name of Flag	Possible Values of Flag	Results of Conversion
	`mwArrayFormatCell`	MATLAB cell array from general `Variant` data.
	Array data from an Excel `range` is coded in Visual Basic as an array of `Variant`. Since MATLAB functions typically have matrix arguments, using the default setting makes sense when you are dealing with data from Excel.	
`OutputArrayFormat`	`mwArrayFormatAsIs`	Array of `Variant`
	Converts arrays according to the default conversion rules listed in MATLAB to COM VARIANT Conversion Rules.	
	`mwArrayFormatMatrix`	A `Variant` containing an array of a basic type.
	`mwArrayFormatCell`	MATLAB cell array from general `Variant` data.
`AutoResizeOutput`	When this flag is set, the target range automatically resizes to fit the resulting array. If this flag is not set, the target range must be at least as large as the output array or the data is truncated. Use this flag for Excel `Range` objects passed directly as output parameters.	
`TransposeOutput`	Transposes all array output. Use this flag when dealing with an encapsulated MATLAB function whose output is a one-dimensional array. By default, the MATLAB product handles one-dimensional arrays as 1-by-*n* matrices (that is, as row vectors). Change this default with the `TransposeOutput` flag if you prefer column output.	

Using Array Formatting Flags

Consider the following Microsoft Visual Basic function definition for `foo`:

```
Sub foo( )
   Dim aClass As mycomponent.myclass
   Dim var1(1 To 2, 1 To 2), var2 As Variant
   Dim x(1 To 2, 1 To 2) As Double
   Dim y1,y2 As Variant

   On Error Goto Handle_Error
```

```
   var1(1,1) = 11#
   var1(1,2) = 12#
   var1(2,1) = 21#
   var1(2,2) = 22#
   x(1,1) = 11
   x(1,2) = 12
   x(2,1) = 21
   x(2,2) = 22
   var2 = x
   Set aClass = New mycomponent.myclass
   Call aClass.foo(1,y1,var1)
   Call aClass.foo(1,y2,var2)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

The example has two `Variant` variables, `var1` and `var2`. These two variables contain the same numerical data, but internally they are structured differently; one is a 2-by-2 array of `variant` and the other is a 1-by-1 array of `variant`. The variables are described in the following table.

Scenario	var1	var2
Numerical data	11 12 21 22	11 12 21 22
Internal structure in Visual Basic	2-by-2 array of `Variant`. Each `variant` is a 1-by-1 array of `Double`.	1-by-1 `Variant`, which contains a 2-by-2 array of `Double`
Result of conversion by the compiler according to the default data conversion rules	2-by-2 cell array. Each element is a 1-by-1 array of `double`.	2-by-2 matrix. Each element is a `Double`.

The `InputArrayFormat` flag controls how the arrays are handled. In this example, the value for the `InputArrayFormat` flag is the default, which is `mwArrayFormatMatrix`. The default causes an array to be converted to a matrix. See the table for the result of the conversion of `var2`.

To specify a cell array (instead of a matrix) as input to the function call, set the `InputArrayFormat` flag to `mwArrayFormatCell` instead of the default. Do this in this example by adding the following line after creating the class and before the method call:

```
aClass .MWFlags.ArrayFormatFlags.InputArrayFormat =
mwArrayFormatCell
```

Setting the flag to `mwArrayFormatCell` causes all array input to the encapsulated MATLAB function to be converted to cell arrays.

Modifying Output Format

Similarly, you can manipulate the format of output arguments using the `OutputArrayFormat` flag. You can also modify array output with the `AutoResizeOutput` and `TransposeOutput` flags.

Output Format in VBScript

When calling a COM object in VBScript you need to make sure that you set `MWFlags` for the COM object to specify cell array for the output. Also, you must use an enumeration (the enumeration value for a cell array is 2) to make the specification (rather than specifying `mwArrayFormatCell`).

The following sample code shows how to accomplish this:

```
obj.MWFlags.ArrayFormatFlags.OutputArrayFormat = 2
```

Using Data Conversion Flags

Two data conversion flags, `CoerceNumericToType` and `InputDateFormat`, govern how numeric and date types are converted from Visual Basic to MATLAB.

This example converts `var1` of type `Variant/Integer` to an `int16` and `var2` of type `Variant/Double` to a `double`.

```
Sub foo( )
   Dim aClass As mycomponent.myclass
   Dim var1, var2 As Variant
   Dim y As Variant

   On Error Goto Handle_Error
   var1 = 1
   var2 = 2#
   Set aClass = New mycomponent.myclass
   Call aClass.foo(1,y,var1,var2)
   Exit Sub
Handle_Error:
```

```
   MsgBox(Err.Description)
End Sub
```

If the original MATLAB function expects `double`s for both arguments, this code might cause an error. One solution is to assign a `double` to `var1`, but this may not be possible or desirable. As an alternative, you can set the `CoerceNumericToType` flag to `mwTypeDouble`, causing the data converter to convert all numeric input to `double`. To do this, place the following line after creating the class and before calling the methods:

```
aClass .MWFlags.DataConversionFlags.CoerceNumericToType =
mwTypeDouble
```

The next example shows how to use the `InputDateFormat` flag, which controls how the Visual Basic `Date` type is converted. The example sends the current date and time as an input argument and converts it to a string.

```
Sub foo( )
   Dim aClass As mycomponent.myclass
   Dim today As Date
   Dim y As Variant

   On Error Goto Handle_Error
   today = Now
   Set aClass = New mycomponent.myclass
   aClass. MWFlags.DataConversionFlags.InputDateFormat =
mwDateFormatString
   Call aClass.foo(1,y,today)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

The next example uses an `MWArg` object to modify the conversion flags for one argument in a method call. In this case the first output argument (`y1`) is coerced to a `Date`, and the second output argument (`y2`) uses the current default conversion flags supplied by `aClass`.

```
Sub foo(y1 As Variant, y2 As Variant)
    Dim aClass As mycomponent.myclass
    Dim ytemp As MWArg
 Dim today As Date

 On Error Goto Handle_Error
 today = Now
```

```
 Set aClass = New mycomponent.myclass
 Set ytemp = New MWArg
 ytemp.MWFlags.DataConversionFlags.OutputAsDate = True
 Call aClass.foo(2, ytemp, y2, today)
 y1 = ytemp
 Exit Sub
Handle_Error:
 MsgBox(Err.Description)
End Sub
```

Special Flags for Some Microsoft Visual Basic Types

In general, you use the `MWFlags` class property to change specified behaviors of the conversion from Microsoft Visual Basic `Variant` types to MATLAB types, and vice versa. There are some exceptions — some types generated by the compiler have their own `MWFlags` property. When you use these particular types, the method call behaves according to the settings of the type and not of the class containing the method being called. The exceptions are for the following types generated by the compiler:

- `MWStruct`
- `MWField`
- `MWComplex`
- `MWSparse`
- `MWArg`

Note: The `MWArg` class is supplied specifically for the case when a particular argument needs different settings from the default class properties.

Use MATLAB Global Variables in Visual Basic

Class properties allow an object to retain an internal state between method calls.

Global variables are variables that are declared in the MATLAB product with the `global` keyword. MATLAB Compiler SDK automatically converts all global variables shared by the MATLAB files that make up a class to properties on that class.

Properties are useful when you have a large array containing values that do not change often, but are operated on frequently. In such cases, setting the array as a property saves the overhead required to pass it to a method every time it is called.

The following example shows how to use a class property in a matrix factorization class. The example develops a class that performs Cholesky, LU, and QR factorizations on the same matrix. It stores the input matrix as a class property so that it is not passed to the factorization routines.

Consider these three MATLAB files.

`Cholesky.m`

```
function [L] = Cholesky()
    global A;
    if (isempty(A))
        L = [];
        return;
    end
    L = chol(A);
```

`LUDecomp.m`

```
function [L,U] = LUDecomp()
    global A;
    if (isempty(A))
        L = [];
        U = [];
        return;
    end
    [L,U] = lu(A);
```

`QRDecomp.m`

```
function [Q,R] = QRDecomp()
    global A;
```

```
    if (isempty(A))
        Q = [];
        R = [];
        return;
    end
    [Q,R] = qr(A);
```

These three files share a common global variable `A`. Each function performs a matrix factorization on `A` and returns the results.

To build the class:

1. Create a compiler project named `mymatrix` with a version of 1.0.
2. Add a single class called `myfactor` to the component.
3. Add the above three MATLAB files to the class.
4. Build the component.

Use the following Visual Basic subroutine to test the `myfactor` class:

```
Sub TestFactor()
    Dim x(1 To 2, 1 To 2) As Double
    Dim C As Variant, L As Variant, U As Variant, _
    Q As Variant, R As Variant
    Dim factor As myfactor

    On Error GoTo Handle_Error
    Set factor = New myfactor
    x(1, 1) = 2#
    x(1, 2) = -1#
    x(2, 1) = -1#
    x(2, 2) = 2#
    factor.A = x
    Call factor.cholesky(1, C)
    Call factor.ludecomp(2, L, U)
    Call factor.qrdecomp(2, Q, R)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Run the subroutine, which does the following:

1. Creates an instance of the `myfactor` class

2. Assigns a double matrix to the property A
3. Calls the three factorization methods

Block Execution of Applications that Create Figures

In this section...
"MCRWaitForFigures" on page 2-20
"Use MCRWaitForFigures to Block Execution" on page 2-20

MCRWaitForFigures

The MATLAB Compiler SDK product adds a `MCRWaitForFigures` method to each class in the COM components that it creates. `MCRWaitForFigures` takes no arguments. Your application can call `MCRWaitForFigures` any time during execution.

The purpose of `MCRWaitForFigures` is to block execution of a calling program as long as figures created in encapsulated MATLAB code are displayed. Typically you use `MCRWaitForFigures` when:

- There are one or more figures open that were created by an instance of a COM object created by the compiler.
- The method that displays the graphics requires user input before continuing.
- The method that calls the figures was called from `main()` in a console program.

When `MCRWaitForFigures` is called, execution of the calling program is blocked if any figures created by the calling object remain open.

Caution Be careful when calling the `MCRWaitForFigures` method. Calling this method from a Microsoft Visual Basic UI or from an interactive program such as Microsoft Excel can hang the application. This method should be called *only* from console-based programs.

Use MCRWaitForFigures to Block Execution

The following example illustrates using `MCRWaitForFigures` from a Microsoft Visual C++ console application. The example uses a COM object created by the compiler; the object encapsulates MATLAB code that draws a simple plot.

1. Create a work folder for your source code. In this example, the folder is `D:\work\plotdemo`.
2. Create the following MATLAB file in this folder:

```
drawplot.m

function drawplot()
    plot(1:10);
```

3 Use the compiler to create a COM component with the following properties:

Component name	`plotdemo`
Class name	`plotdemoclass`
Version	`1.0`

Note: Instead of using the **Library Compiler** app, you can create the component by issuing the following command at the MATLAB prompt:

```
mcc -d 'D:\work\plotdemo\src' -v -B    'ccom:plotdemo,plotdemoclass,1.0'
   'D:\Work\plotdemo\drawplot.m'
```

4 Create a Visual C++ program in a file named `runplot.cpp` with the following code:

```
#include "src\plotdemo_idl.h"
#include "src\plotdemo_idl_i.c"

int main()
{
    // Initialize the COM library
    HRESULT hr = CoInitialize(NULL);
    // Create an instance of the COM object you created
    Iplotdemoclass* pIplotdemoclass = NULL;
    hr = CoCreateInstance(CLSID_plotdemoclass, NULL,
         CLSCTX_INPROC_SERVER, IID_Iplotdemoclass,
         (void **)&pIplotdemoclass);
    // Call the drawplot method
    hr = pIplotdemoclass->drawplot();
    // Block execution until user dismisses the figure window
    hr = pIplotdemoclass->MCRWaitForFigures();
    // Uninitialize COM
    CoUninitialize();
    return 0;
}
```

5 In the MATLAB Command Window, build the application as follows:

```
mbuild runplot.cpp
```

When you run the application, the program displays a plot from 1 to 10 in a MATLAB figure window. The application ends when you dismiss the figure.

Note: To see what happens without the call to `MCRWaitForFigures`. comment out the call, rebuild the application, and run it. In this case, the figure is drawn and is immediately destroyed as the application exits.

MATLAB Runtime Options

When you roll-out a COM component to end users, there are times when you need to specify MATLAB Runtime options to create a log file or improve performance.

Pass these options with `mcc`

What MATLAB Runtime Options are Supported for COM?

- `-nojvm` — Launches the MATLAB Runtime without a Java® Virtual Machine (JVM™). This can improve performance of deployed applications, in some cases.
- `-logfile` — Allows you to specify a log file name.

How Do I Specify MATLAB Runtime Options?

You do this by invoking the following `MWUtil` API calls:

- `Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])`
- `Function IsMCRJVMEnabled() As Boolean`
- `Function IsMCRInitialized() As Boolean`

Share MATLAB Runtime Instances

In this section...
"What Is a Singleton MATLAB Runtime?" on page 2-24 "Advantages and Disadvantages of Using a Singleton" on page 2-24

What Is a Singleton MATLAB Runtime?

You create an instance of the MATLAB Runtime that can be shared among all subsequent class instances within a component. This is commonly called a shared MATLAB Runtime instance or a *Singleton runtime*.

Advantages and Disadvantages of Using a Singleton

In most cases, a singleton MATLAB Runtime will provide many more advantages than disadvantages. Following are examples of when you might and might not create a shared MATLAB Runtime instance.

When You Should Use a Singleton

If you have multiple users running from a specific instance of MATLAB, using a singleton will most likely:

- Utilize system memory more efficiently
- Decrease MATLAB Runtime start-up or initialization time

When You Might Avoid Using a Singleton

Using a singleton may not benefit you if your application uses a large number of global variables. This causes crosstalk.

Obtain Registry Information

When programming with COM components, you might need details about a component. You can use `componentinfo`, which is a MATLAB function, to query the system registry for details about any installed component.

This example queries the registry for a component named `mycomponent` and a version of 1.0. This component has four methods: `mysum`, `randvectors`, `getdates`, and `myprimes`; two properties: `m` and `n`; and one event: `myevent`.

```
Info = componentinfo('mycomponent', 1, 0)

Info =

       Name: 'mycomponent'
      TypeLib: 'mycomponent 1.0 Type Library'
        LIBID: '{3A14AB34-44BE-11D5-B155-00D0B7BA7544}'
    MajorRev: 1
    MinorRev: 0
    FileName: 'D:\Work\ mycomponent\distrib\mycomponent_1_0.dll'
    Interfaces: [1x1 struct]
    CoClasses: [1x1 struct]

Info.Interfaces

ans =

      Name: 'Imyclass'
       IID: '{3A14AB36-44BE-11D5-B155-00D0B7BA7544}'

Info.CoClasses

ans =

          Name: 'myclass'
         CLSID: '{3A14AB35-44BE-11D5-B155-00D0B7BA7544}'
         ProgID: 'mycomponent.myclass.1_0'
   VerIndProgID: 'mycomponent.myclass'
InprocServer32:'D:\Work\mycomponent\distrib\mycomponent_1_0.dll'
       Methods: [1x4 struct]
    Properties: {'m', 'n'}
        Events: [1x1 struct]
```

```
Info.CoClasses.Events.M

ans =

function myevent(x, y)

Info.CoClasses.Methods

ans =

1x4 struct array with fields:
    IDL
    M
    C
    VB

Info.CoClasses.Methods.M

ans =

function [y] = mysum(varargin)

ans =

function [varargout] = randvectors()

ans =

function [x] = getdates(n, inc)

ans =

function [p] = myprimes(n)
```

The returned structure contains fields corresponding to the most important information from the registry and type library for the component.

Handle Errors During a Method Call

If your application generates an error while creating a class instance or during a class method call, the current procedure creates an exception.

Microsoft Visual Basic provides an exception handling capability through the `On Error Goto <label>` statement, in which the program execution jumps to `<label>` when an error occurs. (`<label>` must be located in the same procedure as the `On Error Goto` statement.) All errors in Visual Basic are handled this way, including errors within the MATLAB code that you have encapsulated into a COM object. An exception creates a Visual Basic `ErrObject` object in the current context in a variable called `Err`.

See the Microsoft Visual Basic documentation for a detailed discussion on Visual Basic error handling.

Integrate Magic Square into a COM Application

In this section...
"Overview" on page 2-28
"Creating the MATLAB File" on page 2-28
"Using the Library Compiler App to Create and Build the Project" on page 2-28
"Creating the Microsoft Visual Basic Project" on page 2-29
"Creating the User Interface" on page 2-29
"Creating the Executable in Microsoft Visual Basic" on page 2-31
"Testing the Application" on page 2-31

Overview

This example uses a simple MATLAB file that takes a single input and creates a magic square of that size. It then builds a COM component using this MATLAB file as a class method. Finally, the example shows the integration of this component into a standalone Microsoft Visual Basic application. The application accepts the magic square size as input and displays the matrix in a ListView control box.

Note: ListView is a Windows Form control that displays a list of items with icons. You can use a list view to create a user interface like the right pane of Windows Explorer. See the MSDN Library for more information about Windows Form controls.

Creating the MATLAB File

To get started, create the MATLAB file `mymagic.m` containing the following code:

```
function y = mymagic(x)y = magic(x);
```

Using the Library Compiler App to Create and Build the Project

1. While in MATLAB, open the Library Compiler app.
2. Select **Generic COM Component** as the application type.
3. Add `magicsquare.m` to the list of exported functions.

   `magicsquare.m` is located in the `MagicDemoComp` folder.

4 Click the **Package** button.

Creating the Microsoft Visual Basic Project

Note This procedure assumes that you are using Microsoft Visual Basic 6.0.

1 Start Visual Basic.
2 In the New Project dialog box, select **Installed** > **Templates** > **Other Languages** > **Visual Basic** > **Windows Form Application** as the project type and click **Open**. This creates a new Visual Basic project with a blank form.
3 From the main menu, select **Project** > **References** to open the Project References dialog box.
4 Select **magicdemo 1.0 Type Library** from the list of available components and click **OK**.
5 Returning to the Visual Basic main menu, select **Project** > **Add Component...** to open the Add New Item dialog box.

Creating the User Interface

After you create the project, add a series of controls to the blank form to create a form with the following settings.

Control Type	Control Name	Properties	Purpose
`Frame`	`Frame1`	Caption = `Magic Squares Demo`	Groups controls
`Label`	`Label1`	Caption = `Magic Square Size`	Labels the magic square edit box.
`TextBox`	`edtSize`		Accepts input of magic square size.
`CommandButton`	`btnCreate`	Caption = `Create`	When pressed, creates a new magic square with current size.
`ListView`	`lstMagic`	GridLines = `True` LabelEdit = `lvwManual` View = `lvwReport`	Displays the magic square.

When the form and controls are complete, add the following code to the form. This code references the control and variable names listed above. If you have given different names for any of the controls or any variable, change this code to reflect those differences.

```
Public Class magicvb
    Private sizeMatrix As Double 'Holds current matrix size
    Private theMagic As magicdemo.magicdemoclass 'magic object instance

    Private Sub magicvb_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        'This function is called when the form is loaded.
        'Creates a new magic class instance.
        On Error GoTo Handle_Error
        theMagic = New magicdemo.magicdemoclass
        sizeMatrix = 0
        Exit Sub
Handle_Error:
        MsgBox(Err.Description)
    End Sub


    Private Sub ShowMatrix(matrixMagic As Object)
        'This function populates the ListView with the contents of
        'y. y is assumed to contain a 2D array.
        Dim szSquare As Long
        Dim indxRow As Long
        Dim indxCol As Long
        Dim nLen As Long
        On Error GoTo Handle_Error
        'Get array size
        If IsArray(matrixMagic) Then
            szSquare = UBound(matrixMagic, 1)
        Else
            szSquare = 1
        End If
        lstMagic.Clear()
        lstMagic.Columns.Add("")
        For cIndx = 1 To szSquare
            lstMagic.Columns.Add(CStr(cIndx))
        Next
        lstMagic.View = View.Details
        For indxRow = 1 To szSquare
            Dim item As New ListViewItem(CStr(indxRow))
            For indxCol = 1 To szSquare
                item.SubItems.Add(Format(matrixMagic(indxRow, indxCol)))
            Next
            lstMagic.Items.Add(item)
        Next


        Exit Sub

Handle_Error:
        MsgBox(Err.Description)
    End Sub
```

```
    Private Sub btnCreate_Click(sender As Object, e As EventArgs) Handles btnCreate.Click
        'This function is called when the Create button is pressed.
        'Calls the mymagic method, and displays the magic square.
        Dim matrixMagic As Object
        If sizeMatrix <= 0 Or theMagic Is Nothing Then Exit Sub
        On Error GoTo Handle_Error
        Call theMagic.mymagic(1, matrixMagic, sizeMatrix)
        Call ShowMatrix(matrixMagic)
        Exit Sub
Handle_Error:
        MsgBox(Err.Description)
    End Sub

    Private Sub edtSize_TextChanged(sender As Object, e As EventArgs) Handles edtSize.TextChanged
        'This function is called when ever the contents of the
        'Text box change. Sets the current value of Size.
        On Error Resume Next
        sizeMatrix = CDbl(edtSize.Text)
        If Err.Number > 0 Then
            sizeMatrix = 0
        End If
    End Sub

End Class
```

Creating the Executable in Microsoft Visual Basic

After the code is complete, create the standalone executable `magic.exe`:

1. Reopen the project by selecting **File > Save Project** from the main menu. Accept the default name for the main form and enter `magic.vbp` for the project name.
2. Return to the **File** menu. Select **File > Make magic.exe** to create the finished product.

Testing the Application

You can run the `magic.exe` executable as you would any other program. When the main dialog box opens, enter a positive number in the input box and click **Create**. A magic square of the input size appears.

The `ListView` control automatically implements scrolling if the magic square is larger than 4-by-4.

3

How the MATLAB Compiler SDK Product Creates COM Components

- "Overview of Internal Processes" on page 3-2
- "Component Registration" on page 3-4
- "Data Conversion" on page 3-8
- "Calling Conventions" on page 3-20

Overview of Internal Processes

In this section...
“Code Generation” on page 3-2
“Create Interface Definitions” on page 3-2
“C++ Compilation” on page 3-2
“Linking and Resource Binding” on page 3-3
“Registration of the DLL” on page 3-3

Code Generation

The first step in the build process generates all source code and other supporting files needed to create the component. It also creates the main source file (`mycomponent_dll.cpp`) containing the implementation of each exported function of the DLL. The compiler additionally produces an Interface Description Language (IDL) file (`mycomponent_idl.idl`), containing the specifications for the component's type library, interface, and class, with associated GUIDs. (GUID is an acronym for Globally Unique Identifier, a 128-bit integer guaranteed always to be unique.)

Created next are the C++ class definition and implementation files (`myclass_com.hpp` and `myclass_com.cpp`). In addition to these source files, the compiler generates a DLL exports file (`mycomponent.def`) and a resource script.

Create Interface Definitions

The second step of the build process invokes the IDL compiler on the IDL file generated in step 1 (`mycomponent_idl.idl`), creating the interface header file (`mycomponent_idl.h`), the interface GUID file (`mycomponent_idl_i.c`), and the component type library file (`mycomponent_idl.tlb`). The interface header file contains type definitions and function declarations based on the interface definition in the IDL file. The interface GUID file contains the definitions of the GUIDs from all interfaces in the IDL file. The component type library file contains a binary representation of all types and objects exposed by the component.

C++ Compilation

The third step compiles all C/C++ source files generated in steps 1 and 2 into object code. One additional file containing a set of C++ template classes (`mclcomclass.h`) is

included at this point. This file contains template implementations of all necessary COM base classes, as well as error handling and registration code.

Linking and Resource Binding

The fourth step produces the finished DLL for the component. This step invokes the linker on the object files generated in step 3 and the necessary MATLAB libraries to produce a DLL component (`mycomponent_1_0.dll`). The resource compiler is then invoked on the DLL, along with the resource script generated in step 1, to bind the type library file generated in step 2 into the completed DLL.

Registration of the DLL

The final build step registers the DLL on the system, as described in “Component Registration” on page 3-4.

Component Registration

In this section...

"Self-Registering Components" on page 3-4
"Globally Unique Identifier" on page 3-5
"Versioning" on page 3-6

Self-Registering Components

When the MATLAB Compiler SDK product creates a component, it automatically generates a binary file called a *type library*. As a final step of the build, this file is bound with the resulting DLL as a resource.

MATLAB Compiler SDK COM components are all *self-registering*. A self-registering component contains all the necessary code to add or remove a full description of itself to or from the system registry. The `mwregsvr` utility, distributed with theMATLAB Runtime, registers self-registering DLLs. For example, to register a component called `mycomponent_1_0.dll`, issue this command at the DOS command prompt:

```
mwregsvr mycomponent_1_0.dll
```

When `mwregsvr` completes the registration process, it displays a message indicating success or failure. Similarly, the command

```
mwregsvr /u mycomponent_1_0.dll
```

unregisters the component.

A component installed onto a particular machine must be registered with `mwregsvr`. If you move a component into a different folder on the same machine, you must repeat the registration process. When deleting a component from a specific machine, first unregister it to ensure that the registry does not retain erroneous information.

Tip The `mwregsvr` utility invokes a process that is similar to `regsvr32.exe`, except that `mwregsvr` does not require interaction with a user at the console. The `regsvr32.exe` process belongs to the Windows OS and is used to register dynamic link libraries and Microsoft ActiveX® controls in the registry. This program is important for the stable and secure running of your computer and should not be terminated. You must specify the full path of the component when calling `mwregsvr`, or make the call from the folder in which

the component resides. You can use `regsvr32.exe` as an alternative to `mwregsvr` to register your library.

Globally Unique Identifier

Information is stored in the registry as keys with one or more associated named values. The keys themselves have values of primarily two types: readable strings and GUIDs. (GUID is an acronym for Globally Unique Identifier, a 128-bit integer guaranteed always to be unique.)

The compiler automatically generates GUIDs for COM classes, interfaces, and type libraries that are defined within a component at build time, and codes these keys into the component's self-registration code.

The interface to the system registry is folder based. COM-related information is stored under a top-level key called `HKEY_CLASSES_ROOT`. Under `HKEY_CLASSES_ROOT` are several other keys under which the compiler writes component information.

Caution Do not delete the DLL-file in your project's `src` folder between builds. Doing so causes the GUIDs to be regenerated on the subsequent build. To preserve an older version of the DLL, register it on your system before rebuilding your project.

See the following table for a list of the keys and their definitions.

Key	Definition
`HKEY_CLASSES_ROOT\CLSID`	Information about COM classes on the system. Each component creates a new key under `HKEY_CLASSES_ROOT\CLSID` for each of its COM classes. The key created has a value of the GUID that has been assigned the class and contains several subkeys with information about the class.
`HKEY_CLASSES_ROOT\Interface`	Information about COM interfaces on the system. Each component creates a new key under `HKEY_CLASSES_ROOT\Interface` for each interface it defines.

Key	Definition
	This key has the value of the GUID assigned to the interface and contains subkeys with information about the interface.
`HKEY_CLASSES_ROOT\TypeLib`	Information about type libraries on the system. Each component creates a key for its type library with the value of the GUID assigned to it. Under this key a new key is created for each version of the type library. Therefore, new versions of type libraries with the same name reuse the original GUID but create a new subkey for the new version.
`HKEY_CLASSES_ROOT\<ProgID>`, `HKEY_CLASSES_ROOT\<VerIndProgID>`	These two keys are created for the component's Program ID and Version Independent Program ID. These keys are constructed from strings of the following forms: *`component-name.class-name`* *`component-name.class-name version-number`* These keys are useful for creating a class instance from the component and class names instead of the GUIDs.

Versioning

MATLAB Compiler SDK components support a simple versioning mechanism designed to make building and deploying multiple versions of the same component easy to implement. The version number of a component appears as part of the DLL name, as well as part of the version-dependent ID in the system registry.

When a component is created, you can specify a version number. (The default is 1.0.) During the development of a specific version of a component, the version number should be kept constant. When this is done, the MATLAB Compiler SDK product, in certain cases, reuses type library, class, and interface GUIDs for each subsequent build of the component. This avoids the creation of an excessive number of registry keys for the same component during multiple builds, as occurs if new GUIDs are generated for each build.

When a new version number is introduced, MATLAB Compiler SDK generates new class and interface GUIDs so that the system recognizes them as distinct from previous versions, even if the class name is the same. Therefore, once you deploy a built component, use a new version number for any changes made to the component. This ensures that after you deploy the new component, it is easy to manage the two versions.

MATLAB Compiler SDK implements the versioning rules for a specific component name, class name, and version number by querying the system registry for an existing component with the same name:

- If an existing component has the same version, it uses the GUID of the existing component's type library. If the name of the new class matches the previous version, it reuses the class and interface GUIDs. If the class names do not match, it generates new GUIDs for the new class and interface.
- If it finds an existing component with a different version, it uses the existing type library GUID and creates a new subkey for the new version number. It generates new GUIDs for the new class and interface.
- If it does not find an existing component of the specified name, it generates new GUIDs for the component's type library, class, and interface.

Data Conversion

In this section...
“Conversion Rules” on page 3-8
“Array Formatting Flags” on page 3-17
“Data Conversion Flags” on page 3-18

Conversion Rules

This section describes the data conversion rules for COM components created with the MATLAB Compiler SDK product. These components are dual interface COM objects that support data types compatible with Automation.

Note: *Automation* (formerly called OLE Automation) is a technology that allows software packages to expose their unique features to scripting tools and other applications. Automation uses the Component Object Model (COM), but may be implemented independently from other OLE features, such as in-place activation.

Caution Be aware that IIS (Internet Information Service) usually prevents most COM automation on the basis that it may pose a security risk. Therefore, `XLSREAD` and other Automation services may fail when served by IIS, leading to errors such as `object reference not set`.

When a method is invoked on a MATLAB Compiler SDK component, the input parameters are converted to MATLAB internal array format and passed to the compiled MATLAB function. When the function exits, the output parameters are converted from MATLAB internal array format to COM Automation types.

The COM client passes all input and output arguments in the compiled MATLAB functions as type `VARIANT`. The COM `VARIANT` type is a union of several simple data types. A type `VARIANT` variable can store a variable of any of the simple types, as well as arrays of any of these values.

The Win32 API provides many functions for creating and manipulating `VARIANT`s in C/C++, and Microsoft Visual Basic provides native language support for this type. See the Microsoft Visual Studio documentation for definitions and API support for COM

VARIANTs. VARIANT variables are self describing and store their type code as an internal field of the structure.

Note: This discussion of data refers to both VARIANT and Variant data types. VARIANT is the C++ name and Variant is the corresponding data type in Visual Basic.

See VARIANT Type Codes Supported for a list of the VARIANT type codes supported by compiler components.

See MATLAB to COM VARIANT Conversion Rules and COM VARIANT to MATLAB Conversion Rules for conversion rules between COM VARIANTs and MATLAB arrays.

VARIANT Type Codes Supported

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
VT_EMPTY	-	vbEmpty	-	Uninitialized VARIANT
VT_I1	char	-	-	Signed one-byte character
VT_UI1	unsigned char	vbByte	Byte	Unsigned one-byte character
VT_I2	short	vbInteger	Integer	Signed two-byte integer
VT_UI2	unsigned short	-	-	Unsigned two-byte integer
VT_I4	long	vbLong	Long	Signed four-byte integer
VT_UI4	unsigned long	-	-	Unsigned four-byte integer
VT_R4	float	vbSingle	Single	IEEE® four-byte floating-point value
VT_R8	double	vbDouble	Double	IEEE eight-byte floating-point value
VT_CY	CY+	vbCurrency	Currency	Currency value (64-bit integer, scaled by 10,000)

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
VT_BSTR	BSTR+	vbString	String	String value
VT_ERROR	SCODE+	vbError	-	HRESULT (signed four-byte integer representing a COM error code)
VT_DATE	DATE+	vbDate	Date	Eight-byte floating-point value representing date and time
VT_INT	int	-	-	Signed integer; equivalent to type int
VT_UINT	unsigned int	-	-	Unsigned integer; equivalent to type unsigned int
VT_DECIMAL	DECIMAL+	vbDecimal	-	96-bit (12-byte) unsigned integer, scaled by a variable power of 10
VT_BOOL	VARIANT_BOOL+	vbBoolean	Boolean	Two-byte Boolean value (0xFFFF = True; 0x0000 = False)
VT_DISPATCH	IDispatch*	vbObject	Object	IDispatch* pointer to an object
VT_VARIANT	VARIANT+	vbVariant	Variant	VARIANT (can only be specified if combined with VT_BYREF or VT_ARRAY)
<*anything*>|VT_ARRAY				Bitwise combine VT_ARRAY with any basic type to declare as an array
<*anything*>|VT_BYREF				Bitwise combine VT_BYREF with any

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
				basic type to declare as a reference to a value
[+] Denotes Windows specific type. Not part of standard C/C++.				

MATLAB to COM VARIANT Conversion Rules

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
`cell`	A 1-by-1 cell array converts to a single `VARIANT` with a type conforming to the conversion rule for the MATLAB data type of the cell contents.	A multidimensional cell array converts to a `VARIANT` of type `VT_VARIANT|VT_ARRAY` with the type of each array member conforming to the conversion rule for the MATLAB data type of the corresponding cell.	
`structure`	`VT_DISPATCH`	`VT_DISPATCH`	A MATLAB `struct` array is converted to an `MWStruct` object. (See "Class MWStruct" on page 5-19.) This object is passed as a `VT_DISPATCH` type.
`char`	A 1-by-1 `char` matrix converts to a `VARIANT` of type `VT_BSTR` with string length = 1.	A 1-by-L `char` matrix is assumed to represent a string of length Lin MATLAB. This case converts to a `VARIANT` of type `VT_BSTR` with a string length = L. `char` matrices of more than one row, or of a higher dimensionality convert to a `VARIANT`	Arrays of strings are not supported as `char` matrices. To pass an array of strings, use a cell array of 1-by-L `char` matrices.

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
		of type VT_BSTR|VT_ARRAY. Each string in the converted array is of length 1 and corresponds to each character in the original matrix.	
sparse	VT_DISPATCH	VT_DISPATCH	A MATLAB sparse array is converted to an MWSparse object. (See “Class MWSparse” on page 5-31.) This object is passed as a VT_DISPATCH type.
double	A real 1-by-1 double matrix converts to a VARIANT of type VT_R8. A complex 1-by-1 double matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional double matrix converts to a VARIANT of type VT_R8|VT_ARRAY. A complex multidimensional double matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. See “Class MWComplex” on page 5-28
single	A real 1-by-1 single matrix converts to a VARIANT of type VT_R4. A complex 1-by-1 single matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional single matrix converts to a VARIANT of type VT_R4|VT_ARRAY. A complex multidimensional single matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class.
int8	A real 1-by-1 int8 matrix converts to a VARIANT of type VT_I1. A complex 1-by-1	A real multidimensional int8 matrix converts to a VARIANT of type VT_I1|VT_ARRAY.	Complex arrays are passed to and from compiled MATLAB

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
	int8 matrix converts to a VARIANT of type VT_DISPATCH.	A complex multidimensional int8 matrix converts to a VARIANT of type VT_DISPATCH.	functions using the MWComplex class.
uint8	A real 1-by-1 uint8 matrix converts to a VARIANT of type VT_UI1. A complex 1-by-1 uint8 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional uint8 matrix converts to a VARIANT of type VT_UI1|VT_ARRAY.A complex multidimensional uint8 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class.
int16	A real 1-by-1 int16 matrix converts to a VARIANT of type VT_I2. A complex 1-by-1 int16 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional int16 matrix converts to a VARIANT of type VT_I2|VT_ARRAY. A complex multidimensional int16 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class.
uint16	A real 1-by-1 uint16 matrix converts to a VARIANT of type VT_UI2. A complex 1-by-1 uint16 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional uint16 matrix converts to a VARIANT of type VT_UI2|VT_ARRAY. A complex multidimensional uint16 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class.
int32	A 1-by-1 int32 matrix converts to a VARIANT of type VT_I4. A complex	A multidimensional int32 matrix converts to a VARIANT	Complex arrays are passed to and from compiled MATLAB

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
	1-by-1 `int32` matrix converts to a `VARIANT` of type `VT_DISPATCH`.	of type `VT_I4|VT_ARRAY`. A complex multidimensional `int32` matrix converts to a `VARIANT` of type `VT_DISPATCH`.	functions using the `MWComplex` class.
`uint32`	A 1-by-1 `uint32` matrix converts to a `VARIANT` of type `VT_UI4`. A complex 1-by-1 `uint32` matrix converts to a `VARIANT` of type `VT_DISPATCH`.	A multidimensional `uint32` matrix converts to a `VARIANT` of type `VT_UI4|VT_ARRAY`. A complex multidimensional `uint32` matrix converts to a `VARIANT` of type `VT_DISPATCH`.	Complex arrays are passed to and from compiled MATLAB functions using the `MWComplex` class.
Function handle	`VT_EMPTY`	`VT_EMPTY`	Not supported
Java class	`VT_EMPTY`	`VT_EMPTY`	Not supported
User class	`VT_EMPTY`	`VT_EMPTY`	Not supported
logical	`VT_Bool`	`VT_Bool|VT_ARRAY`	

COM VARIANT to MATLAB Conversion Rules

VARIANT Type	MATLAB Data Type (Scalar or Array Data)	Comments
`VT_EMPTY`	N/A	Empty array created.
`VT_I1`	`int8`	
`VT_UI1`	`uint8`	
`VT_I2`	`int16`	
`VT_UI2`	`uint16`	
`VT_I4`	`int32`	
`VT_UI4`	`uint32`	
`VT_R4`	`single`	

VARIANT Type	MATLAB Data Type (Scalar or Array Data)	Comments
VT_R8	double	
VT_CY	double	
VT_BSTR	char	A VARIANT of type VT_BSTR converts to a 1-by-L MATLAB char array, where L = the length of the string to be converted. A VARIANT of type VT_BSTR|VT_ARRAY converts to a MATLAB cell array of 1-by-L char arrays.
VT_ERROR	int32	
VT_DATE	double	VARIANT dates are stored as doubles starting at midnight Dec. 31, 1899. MATLAB dates are stored as doubles starting at 0/0/00 00:00:00. Therefore, a VARIANT date of 0.0 maps to a MATLAB numeric date of 693960.0. VARIANT dates are converted to MATLAB double types and incremented by 693960.0. VARIANT dates can be optionally converted to strings. See “Data Conversion Flags” on page 3-18 for more information on type coercion.
VT_INT	int32	
VT_UINT	uint32	
VT_DECIMAL	double	
VT_BOOL	logical	
VT_DISPATCH	*Varies*	IDispatch* pointers are treated within the context of what they point to. Objects must be supported types with known

VARIANT Type	MATLAB Data Type (Scalar or Array Data)	Comments
		data extraction and conversion rules, or expose a generic `Value` property that points to a single `VARIANT` type. Data extracted from an object is converted based on the rules for the particular `VARIANT` obtained. Currently, support exists for Excel `Range` objects as well as the types `MWStruct`, `MWComplex`, `MWSparse`, and `MWArg`. See "Reference Utility Classes" on page 5-2 for information on the types to use with COM components.
anything|`VT_BYREF`	*Varies*	Pointers to any of the basic types are processed according to the rules for what they point to. The resulting MATLAB array contains a deep copy of the values.
anything|`VT_ARRAY`	*Varies*	Multidimensional `VARIANT` arrays convert to multidimensional MATLAB arrays, each element converted according to the rules for the basic types. Multidimensional `VARIANT` arrays of type `VT_VARIANT|VT_ARRAY` convert to multidimensional cell arrays, each cell converted according to the rules for that specific type.

Array Formatting Flags

The components have flags that control how array data is formatted in both directions. Generally, you should develop client code that matches the intended inputs and outputs of the MATLAB functions with the corresponding methods on the compiled COM objects, in accordance with the rules listed in MATLAB to COM VARIANT Conversion Rules and COM VARIANT to MATLAB Conversion Rules. In some cases this is not possible, for example, when existing MATLAB code is used in conjunction with a third-party product like Excel.

The following table shows the array formatting flags.

Array Formatting Flags

Flag	Description
`InputArrayFormat`	Defines the array formatting rule used on input arrays. An input array is a `VARIANT` array, created by the client, sent as an input parameter to a method call on a compiled COM object. Valid values for this flag are `mwArrayFormatAsIs`, `mwArrayFormatMatrix`, and `mwArrayFormatCell`. `mwArrayFormatAsIs` passes the array unchanged. `mwArrayFormatMatrix` (default) formats all arrays as matrices. When the input `VARIANT` is of type `VT_ARRAY|`*`type`*, where *`type`* is any numeric type, this flag has no effect. When the input `VARIANT` is of type `VT_VARIANT|VT_ARRAY`, `VARIANT`s in the array are examined. If they are single-valued and homogeneous in type, a MATLAB matrix of the appropriate type is produced instead of a cell array. `mwArrayFormatCell` interprets all arrays as MATLAB cell arrays.
`InputArrayIndFlag`	Sets the input array indirection level used with the `InputArrayFormat` flag (applicable only to nested arrays, i.e., `VARIANT` arrays of `VARIANT`s, which themselves are arrays). The default value for this flag is zero, which applies the `InputArrayFormat` flag to the outermost

Flag	Description
	array. When this flag is greater than zero, e.g., equal to N, the formatting rule attempts to apply itself to the Nth level of nesting.
`OutputArrayFormat`	Defines the array formatting rule used on output arrays. An output array is a MATLAB array, created by the compiled COM object, sent as an output parameter from a method call to the client. The values for this flag, `mwArrayFormatAsIs`, `mwArrayFormatMatrix`, and `mwArrayFormatCell`, cause the same behavior as the corresponding `InputArrayFormat` flag values.
`OutputArrayIndFlag`	(Applies to nested cell arrays only.) Output array indirection level used with the `OutputArrayFormat` flag. This flag works exactly like `InputArrayIndFlag`.
`AutoResizeOutput`	(Applies to Excel ranges only.) When the target output from a method call is a range of cells in an Excel worksheet and the output array size and shape is not known at the time of the call, set this flag to `True` to resize each Excel range to fit the output array.
`TransposeOutput`	Set this flag to `True` to transpose the output arguments. Useful when calling a component from Excel where the MATLAB function returns outputs as row vectors, and you want the data in columns.

Data Conversion Flags

MATLAB Compiler SDK components contain flags to control the conversion of certain `VARIANT` types to MATLAB types. These flags are as follows:

- "CoerceNumericToType" on page 3-18
- "InputDateFormat" on page 3-19
- "OutputAsDate As Boolean" on page 3-19
- "DateBias As Long" on page 3-19

CoerceNumericToType

This flag tells the data converter to convert all numeric `VARIANT` data to one specific MATLAB type. `VARIANT` type codes affected by this flag are `VT_I1`, `VT_UI1`, `VT_I2`,

VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R8, VT_CY, VT_DECIMAL, VT_INT, VT_UINT, VT_ERROR, VT_BOOL, and VT_DATE. Valid values for this flag are mwTypeDefault, mwTypeChar, mwTypeDouble, mwTypeSingle, mwTypeLogical, mwTypeInt8, mwTypeUint8, mwTypeInt16, mwTypeUint16, mwTypeInt32, and mwTypeUint32.

The default for this flag, mwTypeDefault, converts numeric data according to the rules listed in "Data Conversion" on page 3-8.

InputDateFormat

This flag tells the data converter how to convert VARIANT dates to MATLAB dates. Valid values for this flag are mwDateFormatNumeric (default) and mwDateFormatString. The default converts VARIANT dates according to the rule listed in VARIANT Type Codes Supported . The mwDateFormatString flag converts a VARIANT date to its string representation. This flag only affects VARIANT type code VT_DATE.

OutputAsDate As Boolean

This flag instructs the data converter to process an output argument as a date. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as Doubles that need to be decremented by the COM date bias (693960) as well as coerced to COM dates. Set this flag to True to convert all output values of type Double.

DateBias As Long

This flag sets the date bias for performing COM to MATLAB numeric date conversions. The default value of this property is 693960, which represents the difference between the COM Date type and MATLAB numeric dates. This flag allows existing MATLAB code that already performs the increment of numeric dates by 693960 to be used unchanged with the components. To process dates with such code, set this property to 0.

Calling Conventions

In this section...
“Producing a COM Class” on page 3-20
“IDL Mapping” on page 3-21
“Microsoft Visual Basic Mapping” on page 3-22

Producing a COM Class

Producing a COM class requires the generation of

- A class definition file in Interface Description Language (IDL)
- One or more associated C++ class definition/implementation files

The MATLAB Compiler SDK product automatically produces the necessary IDL and C/C ++ code to build each COM class in the component. This process is generally transparent to you when you use the compiler to generate a COM component, and to users of the COM component when they program with it.

For information about IDL and C++ coding rules for building COM objects and for mappings to other languages, see articles in the MSDN Library.

The following table shows the mapping of a generic MATLAB function to IDL code and to Microsoft Visual Basic.

Code	Sample
Generic MATLAB Code	`function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)`

IDL Code	`HRESULT foo([in] long nargout,`
	`  [in,out] VARIANT* Y1,`
	`  [in,out] VARIANT* Y2,`
	`  .`
	`  .`
	`  [in,out] VARIANT* varargout,`
	`  [in] VARIANT X1,`
	`  [in] VARIANT X2,`
	`  .`
	`  .`
	`  [in] VARIANT varargin);`

Visual Basic Code	`Sub foo(nargout As Long, _`
	`  Y1 As Variant, _`
	`  Y2 As Variant, _`
	`  .`
	`  .`
	`  varargout As Variant, _`
	`  X1 As Variant, _`
	`  X2 As Variant, _`
	`  .`
	`  .`
	`  varargin As Variant)`

IDL Mapping

The IDL function definition is generated by producing a function with the same name as the original MATLAB function and an argument list containing all inputs and outputs of the original plus one additional parameter, `nargout`.

When present, the `nargout` parameter is an `[in]` parameter of type `long`. It is always the first argument in the list. This parameter allows correct passage of the MATLAB `nargout` parameter to the compiled MATLAB code. The `nargout` parameter is not produced if you encapsulate a MATLAB function containing no outputs.

Following the `nargout` parameter, the outputs are listed in the order they appear on the left side of the MATLAB function, and are tagged as `[in,out]`, meaning that they are passed in both directions.

The function inputs are listed next, appearing in the same order as they do on the right side of the original function. All inputs are tagged as `[in]` parameters.

When present, the optional `varargin`/`varargout` parameters are always listed as the last input parameters and the last output parameters. All parameters other than `nargout` are passed as COM `VARIANT` types. “Data Conversion” on page 3-8 lists the rules for conversion between MATLAB arrays and COM `VARIANT`s.

Microsoft Visual Basic Mapping

Microsoft Visual Basic provides native support for COM `Variant`s with the `Variant` type, as well as implicit conversions for all Visual Basic primitive types to and from `Variant`s. In general, arrays/scalars of any Visual Basic primitive type, as well as arrays/scalars of Variant types, can be passed as arguments.

MATLAB Compiler SDK COM components also provide direct support for the Excel `Range` object, used by Visual Basic for Applications to represent a range of cells in an Excel worksheet.

See the Visual Basic for Applications documentation included with Microsoft Excel for more information on Visual Basic data types.

See the MSDN Library for more information about Visual Basic and about Excel `Range` manipulation.

Distribute Integrated COM Applications

- "Package COM Applications" on page 4-2
- "About the MATLAB Runtime" on page 4-3
- "Download the MATLAB Runtime Installer" on page 4-5
- "Install the MATLAB Runtime" on page 4-6
- "MATLAB and MATLAB Runtime on Same Machine" on page 4-10
- "Multiple MATLAB Runtime Versions on Single Machine" on page 4-11
- "Uninstall MATLAB Runtime" on page 4-12

Package COM Applications

1. Gather and package the following files for installation on end user computers:
   - MATLAB Runtime installer

     See “Download the MATLAB Runtime Installer” on page 4-5.
   - MATLAB generated COM component
   - `mwcomutil.dll`
   - Executable for the application
2. Include directions for installing the MATLAB Runtime.

   See “Install the MATLAB Runtime” on page 4-6.
3. Include directions for registering `mwcomutil.dll` and the generated component,

About the MATLAB Runtime

In this section...
"How is the MATLAB Runtime Different from MATLAB?" on page 4-3
"Performance Considerations and the MATLAB Runtime" on page 4-4

The MATLAB Runtime is a standalone set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use artifacts built with MATLAB Compiler SDK require access to an appropriate version of the MATLAB Runtime to run.

End-users of compiled artifacts without access to MATLAB must install the MATLAB Runtime on their computers or know the location of a network-installed MATLAB Runtime. The installers generated by the compiler apps may include the MATLAB Runtime installer. If you compiled your artifact using `mcc`, you should direct your end-users to download the MATLAB Runtime installer from the website http://www.mathworks.com/products/compiler/mcr.

See "Install the MATLAB Runtime" on page 4-6 for more information.

How is the MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In the MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. The MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 4.10 (R2009a) of MATLAB Compiler, users who do not have MATLAB installed must have version 7.10 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MATLAB Runtime

MATLAB Compiler SDK was designed to work with a large range of applications that use the MATLAB programming language. Because of this, run-time libraries are large.

Since the MATLAB Runtime technology provides full support for the MATLAB language, including the Java programming language, starting a compiled application takes approximately the same amount of time as starting MATLAB. The amount of resources consumed by the MATLAB Runtime is necessary in order to retain the power and functionality of a full version of MATLAB.

Calls into the MATLAB Runtime are serialized so calls into the MATLAB Runtime are threadsafe. This can impact performance.

Download the MATLAB Runtime Installer

Download the MATLAB Runtime from the website at http://www.mathworks.com/products/compiler/mcr.

Install the MATLAB Runtime

In this section...
“Install the MATLAB Runtime Interactively” on page 4-6 “Install the MATLAB Runtime Non-Interactively” on page 4-7

Install the MATLAB Runtime Interactively

To install the MATLAB Runtime:

1 Start the MATLAB Runtime installer.

Computer	Steps
Windows	Double-click the compiled MATLAB code package self-extracting archive file, typically named *my_program*_pkg.exe, where *my_program* is the name of the MATLAB code. This extracts the MATLAB Runtime installer from the archive, along with all the files that make up the MATLAB Runtime. Once all the files have been extracted, the MATLAB Runtime installer starts automatically.

Linux®
Mac
	Extract the contents of the compiled package, which is a Zip file on Linux systems, typically named, *my_program*_pkg.zip, where *my_program* is the name of the compiled MATLAB code. Use the unzip command to extract the files from the package.
	`unzip MCRInstaller.zip`
	Run the MATLAB Runtime installer script, from the directory where you unzipped the package file, by entering:
	`./install`
	For example, if you unzipped the package and MATLAB Runtime installer in \home\USER, you run the ./install from \home\USER.
	Note: On Mac systems, you may need to enter an administrator username and password after you run ./install.

2. When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click **Next** to proceed with the installation.
3. Specify the folder in which you want to install the MATLAB Runtime in the **Folder Selection** dialog box.

   Note: On Windows systems, you can have multiple versions of the MATLAB Runtime on your computer but only one installation for any particular version. If you already have an existing installation, the MATLAB Runtime installer does not display the **Folder Selection** dialog box because you can only overwrite the existing installation in the same folder.
4. Confirm your choices and click **Next**.

   The MATLAB Runtime installer starts copying files into the installation folder.
5. On Linux and Mac systems, after copying files to your disk, the MATLAB Runtime installer displays the **Product Configuration Notes** dialog box. This dialog box contains information necessary for setting your path environment variables. Copy the path information from this dialog box and then click **Next**.
6. Click **Finish** to exit the installer.

Install the MATLAB Runtime Non-Interactively

To install the MATLAB Runtime without having to interact with the installer dialog boxes, use one of the MATLAB Runtime installer's non-interactive modes:

- silent—the installer runs as a background task and does not display any dialog boxes
- automated—the installer displays the dialog boxes but does not wait for user interaction

When run in silent or automated mode, the MATLAB Runtime installer uses default values for installation options. You can override these defaults by using MATLAB Runtime installer command-line options or an installer control file.

Note: When running in silent or automated mode, the installer overwrites the default installation location.

Running the Installer in Silent Mode

To install the MATLAB Runtime in silent mode:

1 Extract the contents of the MATLAB Runtime installer file to a temporary folder, called `$temp` in this documentation.

Note: On Windows systems, **manually** extract the contents of the installer file.

2 Run the MATLAB Runtime installer, specifying the `-mode` option and `-agreeToLicense yes` on the command line.

Note: On most platforms, the installer is located at the root of the folder into which the archive was extracted. On Windows 64, the installer is located in the archives `bin` folder.

Platform	Command
Windows	`setup -mode silent -agreeToLicense yes`
Linux	`./install -mode silent -agreeToLicense yes`
Mac OS X	`./install -mode silent -agreeToLicense yes`

Note: If you do not include the `-agreeToLicense yes` the installer will not install the MATLAB Runtime.

3 View a log of the installation.

On Windows systems, the MATLAB Runtime installer creates a log file, named `mathworks_`*`username`*`.log`, where *`username`* is your Windows log-in name, in the location defined by your `TEMP` environment variable.

On Linux and Mac systems, the MATLAB Runtime installer displays the log information at the command prompt, unless you redirect it to a file.

Customizing a Non-Interactive Installation

When run in one of the non-interactive modes, the installer will use the default values unless told to do otherwise. Like the MATLAB installer, the MATLAB Runtime installer accepts a number of command line options that modify the default installation properties.

Option	Description
`-destinationFolder`	Specifies where the MATLAB Runtime will be installed.
`-outputFile`	Specifies where the installation log file is written.
`-automatedModeTimeout`	Specifies how long, in milliseconds, that the dialog boxes are displayed when run in automatic mode.
`-inputFile`	Specifies an installer control file with the values for all of the above options.

Note: The MATLAB Runtime installer archive includes an example installer control file called `installer_input.txt`. This file contains all of the options available for a full MATLAB installation. Only the options listed in this section are valid for the MATLAB Runtime installer.

MATLAB and MATLAB Runtime on Same Machine

You do not need to install MATLAB Runtime on your machine if your machine has MATLAB installed. The version of MATLAB should be the same as the version of MATLAB that was used to create the compiled MATLAB code.

You can, however, install the MATLAB Runtime for debugging purposes.

Modifying the Path

If you install MATLAB Runtime on a machine that already has MATLAB on it, you must adjust the library path according to your needs.

- **Windows**

  To run deployed MATLAB code against MATLAB Runtime install, *mcr_root**ver*\runtime\win32|win64 must appear on your system path before *matlabroot*\runtime\win32|win64.

  If *mcr_root**ver*\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB Runtime install area.

  If *matlabroot*\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB installation area.

- **UNIX®**

  To run deployed MATLAB code against MATLAB Runtime on Linux, Linux x86-64, or the <mcr_root>/runtime/<arch> folder must appear on your LD_LIBRARY_PATH before *matlabroot*/runtime/<arch>.

  To run deployed MATLAB code on Mac OS X, the <mcr_root>/runtime folder must appear on your DYLD_LIBRARY_PATH before *matlabroot*/runtime/<arch>.

  To run MATLAB on Mac OS X or Intel® Mac, *matlabroot*/runtime/<arch> must appear on your DYLD_LIBRARY_PATH before the <mcr_root>/bin folder.

Multiple MATLAB Runtime Versions on Single Machine

`MCRInstaller` supports the installation of multiple versions of the MATLAB Runtime on a target machine. This allows applications compiled with different versions of the MATLAB Runtime to execute side by side on the same machine.

If you do not want multiple MATLAB Runtime versions on the target machine, you can remove the unwanted ones. On Windows, run **Add or Remove Programs** from the Control Panel to remove any of the previous versions. On UNIX, you manually delete the unwanted MATLAB Runtime. You can remove unwanted versions before or after installation of a more recent version of the MATLAB Runtime, as versions can be installed or removed in any order.

Uninstall MATLAB Runtime

The method you use to uninstall MATLAB Runtime from your computer varies depending on the type of computer.

Windows

1. Start the uninstaller.

   From the Windows Start menu, search for the **Add or Remove Programs** control panel, and double-click MATLAB Runtime in the list.

   You can also launch the MATLAB Runtime uninstaller from the *mcr_root*\uninstall\bin*arch* folder, where *mcr_root* is your MATLAB Runtime installation folder and *arch* is an architecture-specific folder, such as win64.
2. Select the MATLAB Runtime from the list of products in the Uninstall Products dialog box and
3. Click **Next**.
4. Click **Finish**.

Linux

1. Exit the application.
2. Enter this command at the Linux prompt:

   ```
   rm -rf mcr_root
   ```

   where *mcr_root* represents the name of your top-level MATLAB installation folder.

Mac

- Exit the application.
- Navigate to your MATLAB Runtime installation folder. For example, the installation folder might be named MATLAB_Compiler_Runtime.app in your Applications folder.
- Drag your MATLAB Runtime installation folder to the trash, and then select **Empty Trash** from the Finder menu.

Utility Library for Microsoft COM Components

- "Reference Utility Classes" on page 5-2
- "Class MWUtil" on page 5-3
- "Class MWFlags" on page 5-12
- "Class MWStruct" on page 5-19
- "Class MWField" on page 5-26
- "Class MWComplex" on page 5-28
- "Class MWSparse" on page 5-31
- "Class MWArg" on page 5-35
- "Enum mwArrayFormat" on page 5-37
- "Enum mwDataType" on page 5-38
- "Enum mwDateFormat" on page 5-39

Reference Utility Classes

This section describes the `MWComUtil` library. This library is freely distributable and includes several functions used in array processing, as well as type definitions used in data conversion. This library is contained in the file `mwcomutil.dll`. It must be registered once on each machine that uses Microsoft COM components created by MATLAB Compiler or MATLAB Compiler SDK.

Register the `MWComUtil` library at the DOS command prompt with the command:

```
mwregsvr mwcomutil.dll
```

The `MWComUtil` library includes seven classes and three enumerated types. Before using these types, you must make explicit references to the `MWComUtil` type libraries in the Microsoft Visual Basic IDE.

Note: You must specify the full path of the component when calling `mwregsvr`, or make the call from the folder in which the component resides.

Class MWUtil

The `MWUtil` class contains a set of static utility methods used in array processing and application initialization. This class is implemented internally as a singleton (only one global instance of this class per instance of Microsoft Excel). It is most efficient to declare one variable of this type in global scope within each module that uses it. The methods of `MWUtil` are:

In this section...
"Sub MWInitApplication(pApp As Object)" on page 5-3
"Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])" on page 5-5
"Function IsMCRJVMEnabled() As Boolean" on page 5-6
"Function IsMCRInitialized() As Boolean" on page 5-6
"Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])" on page 5-7
"Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])" on page 5-8
"Sub MWDate2VariantDate(pVar)" on page 5-10

The function prototypes use Visual Basic syntax.

Sub MWInitApplication(pApp As Object)

Initializes the library with the current instance of Microsoft Excel.

Parameters

Argument	Type	Description
`pApp`	`Object`	A valid reference to the current Excel application

Return Value

None.

Remarks

This function must be called once for each session of Excel that uses COM components created by MATLAB Compiler. An error is generated if a method call is made to a member class of any MATLAB Compiler SDK COM component, and the library has not been initialized.

Example

This Visual Basic sample initializes the `MWComUtil` library with the current instance of Excel. A global variable of type `Object` named `MCLUtil` holds an instance of the `MWUtil` class, and another global variable of type `Boolean` named `bModuleInitialized` stores the status of the initialization process. The private subroutine `InitModule()` creates an instance of the `MWComUtil` class and calls the `MWInitApplication` method with an argument of `Application`. Once this function succeeds, all subsequent calls exit without recreating the object.

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

Private Sub InitModule()
    If Not bModuleInitialized Then
        On Error GoTo Handle_Error
        If MCLUtil Is Nothing Then
            Set MCLUtil = CreateObject("MWComUtil.MWUtil")
        End If
        Call MCLUtil.MWInitApplication(Application)
        bModuleInitialized = True
        Exit Sub
Handle_Error:
        bModuleInitialized = False
    End If
End Sub
```

Note: If you are developing concurrently with multiple versions of MATLAB and `MWComUtil.dll`, for example, using this syntax:

```
Set MCLUtil = CreateObject("MWComUtil.MWUtil")
```

requires you to recompile your COM modules every time you upgrade. To avoid this, make your call to the `MWUtil` module version-specific, for example:

```
Set MCLUtil = CreateObject("MWComUtil.MWUtilx.x")
```

where *`x.x`* is the specific version number.

Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])

Start MATLAB Runtime with MATLAB Runtime options. Similar to mclInitializeApplication.

Parameters

Argument	Type	Description
`pApp`	`Object`	A valid reference only when called from an Excel application Non Excel COM clients pass in `Empty`.

Return Value

None.

Remarks

Call this function to pass in MATLAB Runtime options (`nojvm`, `logfile`, etc.). Call this function once per process.

Example

This Visual Basic sample initializes the `MWComUtil` library with the current instance of Excel. A global variable of type `Object` named `MCLUtil` holds an instance of the `MWUtil` class, and another global variable of type `Boolean` named `bModuleInitialized` stores the status of the initialization process. The private subroutine `InitModule()` creates an instance of the `MWComUtil` class and calls the `MWInitApplicationWithMCROptions` method with an argument of `Application` and a string array that contains the options. Once this function succeeds, all subsequent calls exit without recreating the object. When this function successfully executes, the MATLAB Runtime starts up with no JVM and a logfile named `logfile.txt`.

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

Private Sub InitModule()
    If Not bModuleInitialized Then
        On Error GoTo Handle_Error
```

```
        If MCLUtil Is Nothing Then
            Set MCLUtil = CreateObject("MWComUtil.MWUtil")
        End If
    Dim mcrOptions(1 To 3) as String
    mcrOptions(1) = "-nojvm"
    mcrOptions(2) = "-logfile"
    mcrOptions(3) = "logfile.txt"
        Call MCLUtil.MWInitApplicationWithMCROptions(Application, mcrOptions)
        bModuleInitialized = True
        Exit Sub
Handle_Error:
        bModuleInitialized = False
    End If
End Sub
```

Note: If you are not using Excel, pass in `Empty` instead of `Application` to `MWInitApplicationWithMCROptions`.

Function IsMCRJVMEnabled() As Boolean

Returns `true` if MATLAB Runtime is launched with JVM; otherwise returns `false`.

Parameters

None.

Return Value

`Boolean`

Function IsMCRInitialized() As Boolean

Returns `true` if MATLAB Runtime is initialized; otherwise returns `true`

Parameters

None.

Return Value

`Boolean`

Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])

Packs a variable length list of `Variant` arguments into a single `Variant` array. This function is typically used for creating a `varargin` cell from a list of separate inputs. Each input in the list is added to the array only if it is not empty or missing. (In Visual Basic, a missing parameter is denoted by a `Variant` type of `vbError` with a value of `&H80020004`.)

Parameters

Argument	Type	Description
`pVarArg`	`Variant`	Receives the resulting array
`[Var0], [Var1], ...`	`Variant`	Optional list of `Variant`s to pack into the array. From 0 to 32 arguments can be passed.

Return Value

None.

Remarks

This function always frees the contents of `pVarArg` before processing the list.

Example

This example uses `MWPack` in a formula function to produce a `varargin` cell to pass as an input parameter to a method compiled from a MATLAB function with the signature

```
function y = mysum(varargin)
    y = sum([varargin{:}]);
```

The function returns the sum of the elements in `varargin`. Assume that this function is a method of a class named `myclass` that is included in a component named `mycomponent` with a version of 1.0. The Visual Basic function allows up to 10 inputs, and returns the result `y`. If an error occurs, the function returns the error string. This function assumes that `MWInitApplication` has been previously called.

```
Function mysum(Optional V0 As Variant, _
```

```
                Optional V1 As Variant, _
                Optional V2 As Variant, _
                Optional V3 As Variant, _
                Optional V4 As Variant, _
                Optional V5 As Variant, _
                Optional V6 As Variant, _
                Optional V7 As Variant, _
                Optional V8 As Variant, _
                Optional V9 As Variant) As Variant
Dim y As Variant
Dim varargin As Variant
Dim aClass As Object
Dim aUtil As Object

    On Error Goto Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
    Set aUtil = CreateObject("MWComUtil.MWUtil")
    Call aUtil.MWPack(varargin,V0,V1,V2,V3,V4,V5,V6,V7,V8,V9)
    Call aClass.mysum(1, y, varargin)
    mysum = y
    Exit Function
Handle_Error:
    mysum = Err.Description
End Function
```

Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])

Unpacks an array of `Variant`s into individual `Variant` arguments. This function provides the reverse functionality of `MWPack` and is typically used to process a `varargout` cell into individual `Variant`s.

Parameters

Argument	Type	Description
`VarArg`	`Variant`	Input array of `Variant`s to be processed
`nStartAt`	`Long`	Optional starting index (zero-based) in the array to begin processing. Default = `0`.

Argument	Type	Description
bAutoResize	Boolean	Optional auto-resize flag. If this flag is True, any Excel range output arguments are resized to fit the dimensions of the Variant to be copied. The resizing process is applied relative to the upper left corner of the supplied range. Default = False.
[pVar0],[pVar1], ...	Variant	Optional list of Variants to receive the array items contained in VarArg. From 0 to 32 arguments can be passed.

Return Value

None.

Remarks

This function can process a Variant array in one single call or through multiple calls using the nStartAt parameter.

Example

This example uses MWUnpack to process a varargout cell into several Excel ranges, while auto-resizing each range. The varargout parameter is supplied from a method that has been compiled from the MATLAB function.

```
function varargout = randvectors
    for i=1:nargout
        varargout{i} = rand(i,1);
    end
```

This function produces a sequence of nargout random column vectors, with the length of the ith vector equal to i. Assume that this function is included in a class named myclass that is included in a component named mycomponent with a version of 1.0. The Visual Basic subroutine takes no arguments and places the results into Excel columns starting

at A1, B1, C1, and D1. If an error occurs, a message box displays the error text. This function assumes that `MWInitApplication` has been previously called.

```
Sub GenVectors()
    Dim aClass As Object
    Dim aUtil As Object
    Dim v As Variant
    Dim R1 As Range
    Dim R2 As Range
    Dim R3 As Range
    Dim R4 As Range

    On Error GoTo Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
    Set aUtil = CreateObject("MWComUtil.MWUtil")
    Set R1 = Range("A1")
    Set R2 = Range("B1")
    Set R3 = Range("C1")
    Set R4 = Range("D1")
    Call aClass.randvectors(4, v)
    Call aUtil.MWUnpack(v,0,True,R1,R2,R3,R4)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Sub MWDate2VariantDate(pVar)

Converts output dates from MATLAB to `Variant` dates.

Parameters

Argument	Type	Description
`pVar`	`Variant`	`Variant` to be converted

Return Value

None.

Remarks

MATLAB handles dates as double-precision floating-point numbers with 0.0 representing 0/0/00 00:00:00. By default, numeric dates that are output parameters from compiled

MATLAB functions are passed as `Double`s that need to be decremented by the COM date bias as well as coerced to COM dates. The `MWDate2VariantDate` method performs this transformation and additionally converts dates in string form to COM date types.

Example

This example uses `MWDate2VariantDate` to process numeric dates returned from a method compiled from the following MATLAB function.

```
function x = getdates(n, inc)
    y = now;
    for i=1:n
        x(i,1) = y + (i-1)*inc;
    end
```

This function produces an `n`-length column vector of numeric values representing dates starting from the current date and time with each element incremented by `inc` days. Assume that this function is included in a class named `myclass` that is included in a component named `mycomponent` with a version of 1.0. The subroutine takes an Excel range and a `Double` as inputs and places the generated dates into the supplied range. If an error occurs, a message box displays the error text. This function assumes that `MWInitApplication` has been previously called.

```
Sub GenDates(R As Range, inc As Double)
    Dim aClass As Object
    Dim aUtil As Object

    On Error GoTo Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
    Set aUtil = CreateObject("MWComUtil.MWUtil")
    Call aClass.getdates(1, R, R.Rows.Count, inc)
    Call aUtil.MWDate2VariantDate(R)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Class MWFlags

The `MWFlags` class contains a set of array formatting and data conversion flags (See "Rules for Data Conversion Between .NET and MATLAB" for more information on conversion between MATLAB and COM Automation types.) All MATLAB Compiler SDK COM components contain a reference to an `MWFlags` object that can modify data conversion rules at the object level. This class contains these properties and method:

In this section...
"Property ArrayFormatFlags As MWArrayFormatFlags" on page 5-12
"Property DataConversionFlags As MWDataConversionFlags" on page 5-15
"Sub Clone(ppFlags As MWFlags)" on page 5-17

Property ArrayFormatFlags As MWArrayFormatFlags

The `ArrayFormatFlags` property controls array formatting (as a matrix or a cell array) and the application of these rules to nested arrays. The `MWArrayFormatFlags` class is a noncreatable class accessed through an `MWFlags` class instance. This class contains six properties:

- "Property InputArrayFormat As mwArrayFormat" on page 5-12
- "Property InputArrayIndFlag As Long" on page 5-13
- "Property OutputArrayFormat As mwArrayFormat" on page 5-13
- "Property OutputArrayIndFlag As Long" on page 5-14
- "Property AutoResizeOutput As Boolean" on page 5-14
- "Property TransposeOutput As Boolean" on page 5-14

Property InputArrayFormat As mwArrayFormat

This property of type `mwArrayFormat` controls the formatting of arrays passed as input parameters to MATLAB Compiler SDK class methods. The default value is `mwArrayFormatMatrix`. The behaviors indicated by this flag are listed in the next table.

Array Formatting Rules for Input Arrays

Value	Behavior
`mwArrayFormatAsIs`	Converts arrays according to the default conversion rules listed in "Rules for Data Conversion Between .NET and MATLAB".
`mwArrayFormatCell`	Coerces all arrays into cell arrays. Input scalar or numeric array arguments are converted to cell arrays with each cell containing a scalar value for the respective index.
`mwArrayFormatMatrix`	Coerces all arrays into matrices. When an input argument is encountered that is an array of `Variant`s (the default behavior is to convert it to a cell array), the data converter converts this array to a matrix if each `Variant` is single valued, and all elements are homogeneous and of a numeric type. If this conversion is not possible, creates a cell array.

Property InputArrayIndFlag As Long

This property governs the level at which to apply the rule set by the `InputArrayFormat` property for nested arrays (an array of `Variant`s is passed and each element of the array is an array itself). It is not necessary to modify this flag for `varargin` parameters. The data conversion code automatically increments the value of this flag by 1 for `varargin` cells, thus applying the `InputArrayFormat` flag to each cell of a `varargin` parameter. The default value is `0`.

Property OutputArrayFormat As mwArrayFormat

This property of type `mwArrayFormat` controls the formatting of arrays passed as output parameters to class methods. The default value is `mwArrayFormatAsIs`. The behaviors indicated by this flag are listed in the next table.

Array Formatting Rules for Output Arrays

Value	Behavior
`mwArrayFormatAsIs`	Converts arrays according to the default conversion rules listed in "Rules for Data Conversion Between .NET and MATLAB".
`mwArrayFormatMatrix`	Coerces all arrays into matrices. When an output cell array argument is encountered (the default behavior converts it to an array of `Variant`s), the data converter converts this array to a `Variant` that contains a simple numeric array if each cell is single valued, and all elements are homogeneous and of a numeric type. If this conversion is not possible, an array of `Variant`s is created.
`mwArrayFormatCell`	Coerces all output arrays into arrays of `Variant`s. Output scalar or numeric array arguments are converted to arrays of `Variant`s, each `Variant` containing a scalar value for the respective index.

Property OutputArrayIndFlag As Long

This property is similar to the `InputArrayIndFalg` property, as it governs the level at which to apply the rule set by the `OutputArrayFormat` property for nested arrays. As with the input case, this flag is automatically incremented by 1 for a `varargout` parameter. The default value of this flag is `0`.

Property AutoResizeOutput As Boolean

This flag applies to Excel ranges only. When the target output from a method call is a range of cells in an Excel worksheet, and the output array size and shape is not known at the time of the call, setting this flag to `True` instructs the data conversion code to resize each Excel range to fit the output array. Resizing is applied relative to the upper left corner of each supplied range. The default value for this flag is `False`.

Property TransposeOutput As Boolean

Setting this flag to `True` transposes the output arguments. This flag is useful when processing an output parameter from a method call on a COM component, where the

MATLAB function returns outputs as row vectors, and you desire to place the data into columns. The default value for this flag is `False`.

Property DataConversionFlags As MWDataConversionFlags

The `DataConversionFlags` property controls how input variables are processed when type coercion is needed. The `MWDataConversionFlags` class is a noncreatable class accessed through an `MWFlags` class instance. This class contains these properties:

- "Property CoerceNumericToType As mwDataType" on page 5-15
- "PropertyDateBias As Long" on page 5-15
- "Property InputDateFormat As mwDateFormat" on page 5-16
- "PropertyOutputAsDate As Boolean" on page 5-17
- "ReplaceMissing As mwReplaceMissingData" on page 5-17

Property CoerceNumericToType As mwDataType

This property converts all numeric input arguments to one specific MATLAB type. This flag is useful is when variables maintained within the Visual Basic code are different types, e.g., `Long`, `Integer`, etc., and all variables passed to the compiled MATLAB code must be doubles. The default value for this property is `mwTypeDefault`, which uses the default rules in "Rules for Data Conversion Between .NET and MATLAB".

PropertyDateBias As Long

This property sets the date bias for performing COM to MATLAB numeric date conversions. The default value of this property is 693960, representing the difference between the COM `Date` type and MATLAB numeric dates. This flag allows existing MATLAB code that already performs the increment of numeric dates by 693960 to be used unchanged with COM components. To process dates with such code, set this property to `0`.

This example uses data conversion flags to reshape the output from a method compiled from a MATLAB function that produces an output vector of unknown length.

```
function p = myprimes(n)
if length(n)~=1, error('N must be a scalar'); end
if n < 2, p = zeros(1,0); return, end
p = 1:2:n;
q = length(p);
p(1) = 2;
```

```
for k = 3:2:sqrt(n)
    if p((k+1)/2)
        p(((k*k+1)/2):k:q) = 0;
    end
end
p = (p(p>0));
```

This function produces a row vector of all the prime numbers between 0 and `n`. Assume that this function is included in a class named `myclass` that is included in a component named `mycomponent` with a version of 1.0. The subroutine takes an Excel range and a `Double` as inputs, and places the generated prime numbers into the supplied range. The MATLAB function produces a row vector, although you want the output in column format. It also produces an unknown number of outputs, and you do not want to truncate any output. To handle these issues, set the `TransposeOutput` flag and the `AutoResizeOutput` flag to `True`. In previous examples, the Visual Basic `CreateObject` function creates the necessary classes. This example uses an explicit type declaration for the `aClass` variable. As with previous examples, this function assumes that `MWInitApplication` has been previously called.

```
Sub GenPrimes(R As Range, n As Double)
    Dim aClass As mycomponent.myclass

    On Error GoTo Handle_Error
    Set aClass = New mycomponent.myclass
    aClass.MWFlags.ArrayFormatFlags.AutoResizeOutput = True
    aClass.MWFlags.ArrayFormatFlags.TransposeOutput = True
    Call aClass.myprimes(1, R, n)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Property InputDateFormat As mwDateFormat

This property converts dates passed as input parameters to method calls on MATLAB Compiler SDK classes. The default value is `mwDateFormatNumeric`. The behaviors indicated by this flag are shown in the following table.

Conversion Rules for Input Dates

Value	Behavior
`mwDateFormatNumeric`	Convert dates to numeric values as indicated by the rule listed in “Rules

Value	Behavior
	for Data Conversion Between .NET and MATLAB".
`mwDateFormatString`	Convert input dates to strings.

PropertyOutputAsDate As Boolean

This property processes an output argument as a date. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as `Double`s that need to be decremented by the COM date bias (693960) as well as coerced to COM dates. Set this flag to `True` to convert all output values of type `Double`.

ReplaceMissing As mwReplaceMissingData

This property is an enumeration and can have two possible values: `mwReplaceNaN` and `mwReplaceZero`.

To treat empty cells referenced by input parameters as zeros, set the value to `mwReplaceZero`. To treat empty cells referenced by input parameters as `NaN`s (Not a Number), set the value to `mwReplaceNaN`.

By default, the value is `mwReplaceZero`.

Sub Clone(ppFlags As MWFlags)

Creates a copy of an `MWFlags` object.

Parameters

Argument	Type	Description
`ppFlags`	`MWFlags`	Reference to an uninitialized `MWFlags` object that receives the copy

Return Value

None

Remarks

Clone allocates a new `MWFlags` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Class MWStruct

The `MWStruct` class passes or receives a `Struct` type to or from a compiled class method. This class contains seven properties/methods:

In this section...
“Sub Initialize([varDims], [varFieldNames])” on page 5-19
“Property Item([i0], [i1], ..., [i31]) As MWField” on page 5-20
“Property NumberOfFields As Long” on page 5-23
“Property NumberOfDims As Long” on page 5-23
“Property Dims As Variant” on page 5-23
“Property FieldNames As Variant” on page 5-23
“Sub Clone(ppStruct As MWStruct)” on page 5-24

Sub Initialize([varDims], [varFieldNames])

This method allocates a structure array with a specified number and size of dimensions and a specified list of field names.

Parameters

Argument	Type	Description
`varDims`	`Variant`	Optional array of dimensions
`varFieldNames`	`Variant`	Optional array of field names

Return Value

None.

Remarks

When created, an `MWStruct` object has a dimensionality of 1-by-1 and no fields. The `Initialize` method dimensions the array and adds a set of named fields to each element. Each time you call `Initialize` on the same object, it is redimensioned. If you do not supply the `varDims` argument, the existing number and size of the array's dimensions unchanged. If you do not supply the `varFieldNames` argument, the existing list of fields is not changed. Calling `Initialize` with no arguments leaves the array unchanged.

Example

The following Visual Basic code illustrates use of the `Initialize` method to dimension struct arrays.

```
Sub foo ()
    Dim x As MWStruct
    Dim y As MWStruct

    On Error Goto Handle_Error
    'Create 1X1 struct arrays with no fields for x, and y
    Set x = new MWStruct
    Set y = new MWStruct

    'Initialize x to be 2X2 with fields "red", "green",
    '                                    and "blue"
    Call x.Initialize(Array(2,2), Array("red", "green", "blue"))
    'Initialize y to be 1X5 with fields "name" and "age"
    Call y.Initialize(5, Array("name", "age"))

    'Re-dimension x to be 3X3 with the same field names
    Call x.Initialize(Array(3,3))

    'Add a new field to y
    Call y.Initialize(, Array("name", "age", "salary"))

    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

Property Item([i0], [i1], ..., [i31]) As MWField

The `Item` property is the default property of the `MWStruct` class. This property is used to set/get the value of a field at a particular index in the structure array.

Parameters

Argument	Type	Description
`i0`,`i1`, ..., `i31`	`Variant`	Optional index arguments. Between 0 and 32 index arguments can be entered. To reference an element of

Argument	Type	Description
		the array, specify all indexes as well as the field name.

Remarks

When accessing a named field through this property, you must supply all dimensions of the requested field as well as the field name. This property always returns a single field value, and generates a bad index error if you provide an invalid or incomplete index list. Index arguments have four basic formats:

- Field name only

  This format may be used only in the case of a 1-by-1 structure array and returns the named field's value. For example:

  ```
  x("red") = 0.2
  x("green") = 0.4
  x("blue") = 0.6
  ```

  In this example, the name of the `Item` property was neglected. This is possible since the `Item` property is the default property of the `MWStruct` class. In this case the two statements are equivalent:

  ```
  x.Item("red") = 0.2
  x("red") = 0.2
  ```

- Single index and field name

This format accesses array elements through a single subscripting notation. A single numeric index `n` followed by the field name returns the named field on the `n`th array element, navigating the array linearly in column-major order. For example, consider a 2-by-2 array of structures with fields `"red"`, `"green"` , and `"blue"` stored in a variable `x`. These two statements are equivalent:

```
y = x(2, "red")
y = x(2, 1, "red")
```

- All indices and field name

This format accesses an array element of an multidimensional array by specifying `n` indices. These statements access all four of the elements of the array in the previous example:

```
For I From 1 To 2
       For J From 1 To 2
              r(I, J) = x(I, J, "red")
              g(I, J) = x(I, J, "green")
              b(I, J) = x(I, J, "blue")
       Next
Next
```

- Array of indices and field name

This format accesses an array element by passing an array of indices and a field name. The next example rewrites the previous example using an index array:

```
Dim Index(1 To 2) As Integer

For I From 1 To 2
       Index(1) = I
       For J From 1 To 2
              Index(2) = J
              r(I, J) = x(Index, "red")
              g(I, J) = x(Index, "green")
              b(I, J) = x(Index, "blue")
       Next
Next
```

With these four formats, the `Item` property provides a very flexible indexing mechanism for structure arrays. Also note:

- You can combine the last two indexing formats. Several index arguments supplied in either scalar or array format are concatenated to form one index set. The combining stops when the number of dimensions has been reached. For example:

  ```
  Dim Index1(1 To 2) As Integer
  Dim Index2(1 To 2) As Integer

  Index1(1) = 1
  Index1(2) = 1
  Index2(1) = 3
  Index2(2) = 2
  x(Index1, Index2, 2, "red") = 0.5
  ```

The last statement resolves to

```
x(1, 1, 3, 2, 2, "red") = 0.5
```

- The field name must be the last index in the list. The following statement produces an error:

  ```
  y = x("blue", 1, 2)
  ```

- Field names are case sensitive.

Property NumberOfFields As Long

The read-only `NumberOfFields` property returns the number of fields in the structure array.

Property NumberOfDims As Long

The read-only `NumberOfDims` property returns the number of dimensions in the struct array.

Property Dims As Variant

The read-only `Dims` property returns an array of length `NumberOfDims` that contains the size of each dimension of the struct array.

Property FieldNames As Variant

The read-only `FieldNames` property returns an array of length `NumberOfFields` that contains the field names of the elements of the structure array.

Example

The next Visual Basic code sample illustrates how to access a two-dimensional structure array's fields when the field names and dimension sizes are not known in advance.

```
Sub foo ()
      Dim x As MWStruct
      Dim Dims as Variant
      Dim FieldNames As Variant


      On Error Goto Handle_Error
      '
      '... Call a method that returns an MWStruct in x
```

```
      '
      Dims = x.Dims
      FieldNames = x.FieldNames
      For I From 1 To Dims(1)
            For J From 1 To Dims(2)
                  For K From 1 To x.NumberOfFields
                        y = x(I,J,FieldNames(K))
                        ' ... Do something with y
                  Next
            Next
      Next
Exit Sub
Handle_Error:
      MsgBox(Err.Description)
End Sub
```

Sub Clone(ppStruct As MWStruct)

Creates a copy of an `MWStruct` object.

Parameters

Argument	Type	Description
`ppStruct`	`MWStruct`	Reference to an uninitialized `MWStruct` object to receive the copy

Return Value

None

Remarks

`Clone` allocates a new `MWStruct` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Example

The following Visual Basic example illustrates the difference between assignment and `Clone` for `MWStruct` objects.

```
Sub foo ()
    Dim x1 As MWStruct
    Dim x2 As MWStruct
    Dim x3 As MWStruct

    On Error Goto Handle_Error
    Set x1 = new MWStruct
    x1("name") = "John Smith"
    x1("age") = 35

   'Set reference of x1 to x2
    Set x2 = x1
   'Create new object for x3 and copy contents of x1 into it
    Call x1.Clone(x3)
   'x2's "age" field is
   'also modified 'x3's "age" field unchanged
    x1("age") = 50
        .
        .
        .
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

Class MWField

The `MWField` class holds a single field reference in an `MWStruct` object. This class is noncreatable and contains four properties/methods:

In this section...
“Property Name As String” on page 5-26
“Property Value As Variant” on page 5-26
“Property MWFlags As MWFlags” on page 5-26
“Sub Clone(ppField As MWField)” on page 5-26

Property Name As String

The name of the field (read only).

Property Value As Variant

Stores the field's value (read/write). The `Value` property is the default property of the `MWField` class. The value of a field can be any type that is coercible to a `Variant`, as well as object types.

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular field. Each field in a structure has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppField As MWField)

Creates a copy of an `MWField` object.

Parameters

Argument	Type	Description
`ppField`	`MWField`	Reference to an uninitialized `MWField` object to receive the copy

Return Value

None.

Remarks

`Clone` allocates a new `MWField` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Class MWComplex

The `MWComplex` class passes or receives a complex numeric array into or from a compiled class method. This class contains four properties/methods:

In this section...
“Property Real As Variant” on page 5-28
“Property Imag As Variant” on page 5-28
“Property MWFlags As MWFlags” on page 5-29
“Sub Clone(ppComplex As MWComplex)” on page 5-29

Property Real As Variant

Stores the real part of a complex array (read/write). The `Real` property is the default property of the `MWComplex` class. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to a numeric matrix (no cell data allowed). Valid Visual Basic numeric types for complex arrays include `Byte`, `Integer`, `Long`, `Single`, `Double`, `Currency`, and `Variant/vbDecimal`.

Property Imag As Variant

Stores the imaginary part of a complex array (read/write). The `Imag` property is optional and can be `Empty` for a pure real array. If the `Imag` property is not empty and the size and type of the underlying array do not match the size and type of the `Real` property's array, an error results when the object is used in a method call.

Example

The following Visual Basic code creates a complex array with the following entries:

```
    x = [ 1+i 1+2i
          2+i 2+2i ]
Sub foo()
    Dim x As MWComplex
    Dim rval(1 To 2, 1 To 2) As Double
    Dim ival(1 To 2, 1 To 2) As Double
```

```
    On Error Goto Handle_Error
    For I = 1 To 2
        For J = 1 To 2
                rval(I,J) = I
                ival(I,J) = J
        Next
    Next
    Set x = new MWComplex
    x.Real = rval
    x.Imag = ival
        .
        .
        .
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular complex array. Each `MWComplex` object has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppComplex As MWComplex)

Creates a copy of an `MWComplex` object.

Parameters

Argument	Type	Description
`ppComplex`	`MWComplex`	Reference to an uninitialized `MWComplex` object to receive the copy

Return Value

None

Remarks

`Clone` allocates a new `MWComplex` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Class MWSparse

The `MWSparse` class passes or receives a two-dimensional sparse numeric array into or from a compiled class method. This class has seven properties/methods:

In this section...
“Property NumRows As Long” on page 5-31
“Property NumColumns As Long” on page 5-31
“Property RowIndex As Variant” on page 5-31
“Property ColumnIndex As Variant” on page 5-32
“Property Array As Variant” on page 5-32
“Property MWFlags As MWFlags” on page 5-32
“Sub Clone(ppSparse As MWSparse)” on page 5-32

Property NumRows As Long

Stores the row dimension for the array. The value of `NumRows` must be nonnegative. If the value is zero, the row index is taken from the maximum of the values in the `RowIndex` array.

Property NumColumns As Long

Stores the column dimension for the array. The value of `NumColumns` must be nonnegative. If the value is zero, the row index is taken from the maximum of the values in the `ColumnIndex` array.

Property RowIndex As Variant

Stores the array of row indices of the nonzero elements of the array. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to or be coercible to a numeric matrix of type `Long`. If the value of `NumRows` is nonzero and any row index is greater than `NumRows`, a bad-index error occurs. An error also results if the number of elements in the `RowIndex` array does not match the number of elements in the `Array` property's underlying array.

Property ColumnIndex As Variant

Stores the array of column indices of the nonzero elements of the array. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to or be coercible to a numeric matrix of type `Long`. If the value of `NumColumns` is nonzero and any column index is greater than `NumColumns`, a bad-index error occurs. An error also results if the number of elements in the `ColumnIndex` array does not match the number of elements in the `Array` property's underlying array.

Property Array As Variant

Stores the nonzero array values of the sparse array. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to or be coercible to a numeric matrix of type `Double` or `Boolean`.

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular sparse array. Each `MWSparse` object has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppSparse As MWSparse)

Creates a copy of an `MWSparse` object.

Parameters

Argument	Type	Description
`ppSparse`	`MWSparse`	Reference to an uninitialized `MWSparse` object to receive the copy

Return Value

None.

Remarks

`Clone` allocates a new `MWSparse` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Example

The following Visual Basic sample creates a 5-by-5 tridiagonal sparse array with the following entries:

```
X = [ 2 -1  0   0   0
     -1  2 -1   0   0
      0 -1  2  -1   0
      0  0 -1   2  -1
      0  0  0  -1   2 ]
```

```
Sub foo()
    Dim x As MWSparse
    Dim rows(1 To 13) As Long
    Dim cols(1 To 13) As Long
    Dim vals(1 To 13) As Double
    Dim I As Long, K As Long

    On Error GoTo Handle_Error
    K = 1
    For I = 1 To 4
        rows(K) = I
        cols(K) = I + 1
        vals(K) = -1
        K = K + 1
        rows(K) = I
        cols(K) = I
        vals(K) = 2
        K = K + 1
        rows(K) = I + 1
        cols(K) = I
        vals(K) = -1
        K = K + 1
    Next
    rows(K) = 5
    cols(K) = 5
    vals(K) = 2
    Set x = New MWSparse
```

```
    x.NumRows = 5
    x.NumColumns = 5
    x.RowIndex = rows
    x.ColumnIndex = cols
    x.Array = vals
        .
        .
        .
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Class MWArg

The `MWArg` class passes a generic argument into a compiled class method. This class passes an argument for which the data conversion flags are changed for that one argument. This class has three properties/methods:

In this section...
“Property Value As Variant” on page 5-35
“Property MWFlags As MWFlags” on page 5-35
“Sub Clone(ppArg As MWArg)” on page 5-35

Property Value As Variant

The `Value` property stores the actual argument to pass. Any type that can be passed to a compiled method is valid for this property.

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular argument. Each `MWArg` object has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppArg As MWArg)

Creates a copy of an `MWArg` object.

Parameters

Argument	Type	Description
`ppArg`	`MWArg`	Reference to an uninitialized `MWArg` object to receive the copy

Return Value

None.

Remarks

Clone allocates a new `MWArg` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Enum mwArrayFormat

The `mwArrayFormat` enumeration is a set of constants that denote an array formatting rule for data conversion.

mwArrayFormat Values

Constant	Numeric Value	Description
`mwArrayFormatAsIs`	0	Do not reformat the array.
`mwArrayFormatMatrix`	1	Format the array as a matrix.
`mwArrayFormatCell`	2	Format the array as a cell array.

Enum mwDataType

The `mwDataType` enumeration is a set of constants that denote a MATLAB numeric type.

mwDataType Values

Constant	Numeric Value	MATLAB Type
`mwTypeDefault`	0	Not applicable
`mwTypeLogical`	3	`logical`
`mwTypeChar`	4	`char`
`mwTypeDouble`	6	`double`
`mwTypeSingle`	7	`single`
`mwTypeInt8`	8	`int8`
`mwTypeUint8`	9	`uint8`
`mwTypeInt16`	10	`int16`
`mwTypeUint16`	11	`uint16`
`mwTypeInt32`	12	`int32`
`mwTypeUint32`	13	`uint32`

Enum mwDateFormat

The `mwDateFormat` enumeration is a set of constants that denote a formatting rule for dates.

mwDateFormat Values

Constant	Numeric Value	Description
`mwDateFormatNumeric`	0	Format dates as numeric values
`mwDateFormatString`	1	Format dates as strings

Functions — Alphabetical List

mcrinstaller

Display version and location information for MATLAB Runtime installer corresponding to current platform

Syntax

```
[INSTALLER_PATH, MAJOR, MINOR, PLATFORM, LIST] = mcrinstaller;
```

Description

Displays information about available MATLAB Runtime installers using the format: `[INSTALLER_PATH, MAJOR, MINOR, PLATFORM, LIST] = mcrinstaller;` where:

- *`INSTALLER_PATH`* is the full path to the installer for the current platform.
- *`MAJOR`* is the major version number of the installer.
- *`MINOR`* is the minor version number of the installer.
- *`PLATFORM`* is the name of the current platform (returned by `COMPUTER(`*`arch`*`)`).
- *`LIST`* is a cell array of strings containing the full paths to MATLAB Runtime installers for other platforms. This list is non-empty only in a multi-platform MATLAB installation.

Note: You must distribute the MATLAB Runtime library to your end users to enable them to run applications developed with MATLAB Compiler or MATLAB Compiler SDK.

See “Install the MATLAB Runtime” for more information about the MATLAB Runtime installer.

Examples

Find MATLAB Runtime Installer Locations

Display locations of MATLAB Runtime installers for platform. This example shows output for a `win64` system.

```
mcrinstaller

The WIN64 MCR Installer, version 7.16, is:
    X:\jobx\clusterc\current\matlab\toolbox\compiler\
                             deploy\win64\MCRInstaller.exe

MCR installers for other platforms are located in:
    X:\jobx\clusterc\current\matlab\toolbox\compiler\
                      deploy\win64
    win64 is the value of COMPUTER(win64) on
           the target machine.

For more information, read your local MCR Installer help.
Or see the online documentation at MathWorks' web site. (Page
                                                 may load slowly.)

ans =

X:\jobx\clusterc\current\matlab\toolbox\compiler\
                             deploy\win64\MCRInstaller.exe
```

mcrversion

Determine version of installed MATLAB Runtime

Syntax

```
[major, minor] = mcrversion;
```

Description

The MATLAB Runtime version number consists of two digits, separated by a decimal point. This function returns each digit as a separate output variable: `[major, minor] = mcrversion;` Major and minor are returned as integers.

If the version number ever increases to three or more digits, call `mcrversion` with more outputs, as follows:

```
[major, minor, point] = mcrversion;
```
At this time, all outputs past “minor” are returned as zeros.

Typing only `mcrversion` will return the major version number only.

Examples

```
mcrversion
ans =
     7
```