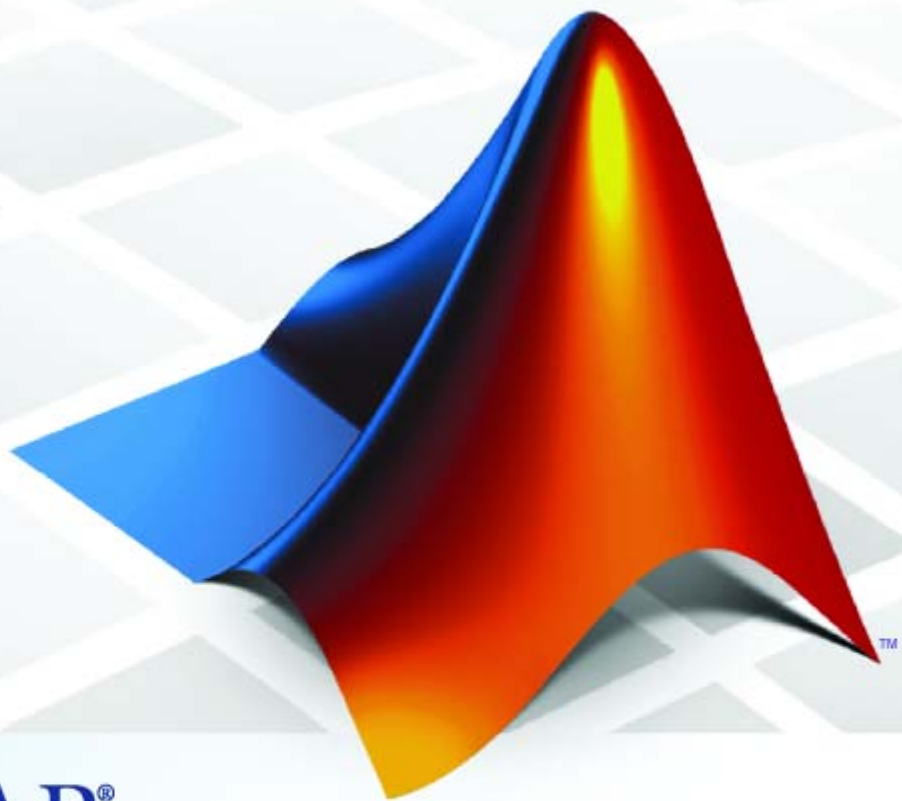


# MATLAB® Builder™ EX 1

## User's Guide



MATLAB®

## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com)  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab)  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html)

Web  
Newsgroup  
Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com)  
[bugs@mathworks.com](mailto:bugs@mathworks.com)  
[doc@mathworks.com](mailto:doc@mathworks.com)  
[service@mathworks.com](mailto:service@mathworks.com)  
[info@mathworks.com](mailto:info@mathworks.com)

Product enhancement suggestions  
Bug reports  
Documentation error reports  
Order status, license renewals, passcodes  
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*MATLAB® Builder™ EX User's Guide*

© COPYRIGHT 1984–2008 The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### Revision History

December 2001	Online only	New for Version 1.0
July 2002	First printing	Revised for Version 1.1 (Release 13)
June 2004	Online only	Revised for Version 1.2 (Release 14) Name changed from MATLAB® MATLAB® Builder™ EX
August 2004	Online only	Revised for Version 1.2.1 (Release 14+)
October 2004	Online only	Revised for Version 1.2.2 (Release 14SP1)
September 2005	Online only	Revised for Version 1.2.5 (Release 14SP3)
March 2006	Online only	Revised for Version 1.2.6 (Release 2006a)
September 2006	Online only	Revised for Version 1.2.7 (Release 2006b)
March 2007	Online only	Revised for Version 1.2.8 (Release 2007a)
September 2007	Online only	Revised for Version 1.2.9 (Release 2007b)
March 2008	Online only	Revised for Version 1.2.10 (Release 2008a)



## Getting Started

### 1

<b>Product Overview</b> .....	<b>1-2</b>
Component Naming Conventions .....	1-2
<b>Building a Deployable Application</b> .....	<b>1-4</b>
Creating and Building a Component .....	1-4
Using the mcc Command to Build a Component .....	1-6
Testing the Component .....	1-6
Deploying the Component .....	1-7
Packaging and Distributing the Component .....	1-8

## Programming with the MATLAB® Builder™ EX Product

### 2

<b>Overview of the Integration Process</b> .....	<b>2-3</b>
<b>When to Use a Formula Function or a Subroutine</b> ....	<b>2-4</b>
<b>Initializing the MATLAB® Builder™ EX Libraries with Excel®</b> .....	<b>2-5</b>
<b>Creating an Instance of a Class</b> .....	<b>2-7</b>
Overview .....	2-7
CreateObject Function .....	2-7
New Operator .....	2-8
How the MCR Is Shared Among Classes .....	2-9
<b>Calling the Methods of a Class Instance</b> .....	<b>2-10</b>
<b>Processing varargin and varargout Arguments</b> .....	<b>2-12</b>

<b>Calling Compiled MATLAB® Functions from Microsoft® Excel®</b> .....	<b>2-14</b>
<b>Handling Errors During a Method Call</b> .....	<b>2-17</b>
<b>Modifying Flags</b> .....	<b>2-18</b>
Overview .....	<b>2-18</b>
Array Formatting Flags .....	<b>2-18</b>
Data Conversion Flags .....	<b>2-21</b>

## Usage Examples

### 3

<b>Magic Square Examples</b> .....	<b>3-2</b>
About the Magic Square Examples .....	<b>3-2</b>
Creating the Project .....	<b>3-3</b>
Adding the MATLAB® Builder™ EX COM Function to Microsoft® Excel® .....	<b>3-3</b>
Illustration 1. Output Magic Square Results to Microsoft® Excel® .....	<b>3-4</b>
Illustration 2. Transpose the Output .....	<b>3-4</b>
Illustration 3. Resize the Output .....	<b>3-5</b>
Inspecting the Microsoft® Visual Basic® Code .....	<b>3-5</b>
<b>Multiple Files and Variable Arguments Example</b> .....	<b>3-6</b>
Overview .....	<b>3-6</b>
Creating the Project .....	<b>3-6</b>
Adding the MATLAB® Builder™ EX COM Function to Microsoft® Excel® .....	<b>3-7</b>
Illustration 4: Calling myplot .....	<b>3-9</b>
Illustration 5: Calling mysum Four Different Ways .....	<b>3-10</b>
Illustration 6: myprimes Macro .....	<b>3-11</b>
Inspecting the Microsoft® Visual Basic® Code .....	<b>3-12</b>
<b>Spectral Analysis Example</b> .....	<b>3-13</b>
Overview .....	<b>3-13</b>
Building the Component .....	<b>3-14</b>
Integrating the Component Using VBA .....	<b>3-15</b>
Testing the Add-In .....	<b>3-27</b>

## Function Wizard

# 4

<b>Overview of the Function Wizard</b> .....	<b>4-2</b>
<b>Installing the Function Wizard Add-In</b> .....	<b>4-3</b>
Installing with Versions of Office Older Than 2007 .....	<b>4-3</b>
Installing with Office 2007 .....	<b>4-3</b>
<b>Starting the Function Wizard</b> .....	<b>4-5</b>
Starting the Function Wizard with Versions of Microsoft® Office Older Than 2007 .....	<b>4-5</b>
Starting the Function Wizard with Office 2007 .....	<b>4-5</b>
<b>Understanding the Function Viewer</b> .....	<b>4-6</b>
Using the Function Viewer .....	<b>4-6</b>
Loading and Executing Functions .....	<b>4-6</b>
<b>Component Browser</b> .....	<b>4-8</b>
<b>Function Properties</b> .....	<b>4-9</b>
The Function Properties Dialog Box .....	<b>4-9</b>
Editing Function Arguments .....	<b>4-10</b>
<b>Argument Properties</b> .....	<b>4-14</b>
Input Argument Properties Dialog Box .....	<b>4-14</b>
Output Argument Properties Dialog Box .....	<b>4-15</b>
<b>Function Utilities</b> .....	<b>4-16</b>
Rename Function Dialog Box .....	<b>4-16</b>
Copy Function Dialog Box .....	<b>4-16</b>
Move Function Dialog Box .....	<b>4-17</b>

5

Utility Library for Microsoft® COM Components

6

**Referencing the Utility Classes** ..... 6-2

**Utility Library Classes** ..... 6-3

- Class MWUtil ..... 6-3
- Class MWFlags ..... 6-10
- Class MWStruct ..... 6-16
- Class MWField ..... 6-23
- Class MWComplex ..... 6-24
- Class MWSparse ..... 6-26
- Class MWArg ..... 6-29

**Enumerations** ..... 6-31

- Enum mwArrayFormat ..... 6-31
- Enum mwDataType ..... 6-31
- Enum mwDateFormat ..... 6-32

Data Conversion

A

**Data Conversion Rules** ..... A-2

**Array Formatting Flags** ..... A-12

**Data Conversion Flags** ..... A-14

- CoerceNumericToType ..... A-14
- InputDateFormat ..... A-15
- OutputAsDate As Boolean ..... A-16
- DateBias As Long ..... A-16



## Utility Library

### B

<b>Referencing Utility Classes</b> .....	<b>B-2</b>
<b>Utility Library Classes</b> .....	<b>B-3</b>
Class MWUtil .....	<b>B-3</b>
Class MWFlags .....	<b>B-10</b>
Class MWStruct .....	<b>B-16</b>
Class MWField .....	<b>B-24</b>
Class MWComplex .....	<b>B-25</b>
Class MWSparse .....	<b>B-27</b>
Class MWArg .....	<b>B-30</b>
<b>Enumerations</b> .....	<b>B-32</b>
Enum mwArrayFormat .....	<b>B-32</b>
Enum mwDataType .....	<b>B-32</b>
Enum mwDateFormat .....	<b>B-33</b>

## Troubleshooting

### C

### Examples

### D

<b>Calling a MATLAB Function from Microsoft® Excel®</b> ..	<b>D-2</b>
<b>Using Multiple Files and Variable Arguments</b> .....	<b>D-2</b>
<b>Creating a Comprehensive Microsoft® Excel® Add-In: Spectral Analysis</b> .....	<b>D-2</b>



# Getting Started

---

Product Overview (p. 1-2)

Summarizes the MATLAB®  
Builder™ EX product

Building a Deployable Application  
(p. 1-4)

Describes how to create and deploy  
an application

## Product Overview

The MATLAB® Builder™ EX product is an extension to the MATLAB® Compiler™ product. You use the builder to package MATLAB® functions so that Microsoft® Excel® users can access them from Excel®.

The builder converts MATLAB M-functions to methods of a class that you define. From this class, the builder creates *components*. The MATLAB Builder EX product components are Microsoft® COM objects that are accessible from Microsoft Excel through Visual Basic® for Applications (VBA).

COM is an acronym for Component Object Model, which is a Microsoft binary standard for object interoperability. COM components use a common integration architecture that provides a consistent model across multiple applications. All Microsoft Office XP applications support COM add-ins.

Each COM object exposes a *class* to the Visual Basic programming environment. The class contains a set of functions called methods. These methods correspond to the original MATLAB functions included in the component's project. The COM components created by the MATLAB Builder EX product contain one or more classes, and each class provides an interface to the M-functions that you add to the class at build time. The COM component provides a set of methods that wrap the M-code along with a DLL file.

---

**Note** Currently, the MATLAB Builder EX components support one class per component.

---

When you package and distribute an application that uses your component, you must include supporting files generated by the MATLAB Builder EX product as well as the MATLAB Component Runtime (MCR), which gives you access to an entire library of MATLAB functions within one file.

## Component Naming Conventions

When creating a component, you must additionally provide a class name. The component name represents the name of the Dynamic Load Library (DLL) file to be created. The class name denotes the name of the class that performs a

call on a specific method at run time. The relationship between component name and class name, and which methods (MATLAB functions) go into a particular class, are purely organizational.

As a general rule, when compiling many MATLAB functions, it helps to determine a scheme of function categories and to create a separate class for each category. The name of each class should describe what the class does. Organizing related functions into classes in this way reduces the amount of code to rebuild and redeploy when one function is changed.

Valid characters are any alpha or numeric characters, as well as the underscore ( `_` ) character.

## Building a Deployable Application

In this section...
“Creating and Building a Component” on page 1-4
“Using the mcc Command to Build a Component” on page 1-6
“Testing the Component” on page 1-6
“Deploying the Component” on page 1-7
“Packaging and Distributing the Component” on page 1-8

### Creating and Building a Component

To use the MATLAB® Builder™ EX product to build a component:

- 1** If you have not already done so, enter the following MATLAB® command at the command line:

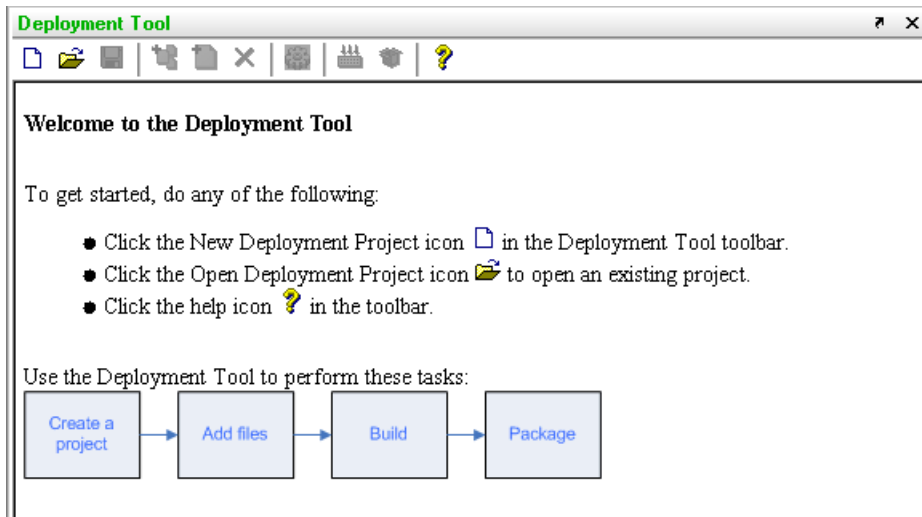
```
mbuild -setup
```

Be sure to choose a supported compiler. See Supported Compilers at <http://www.mathworks.com/support/tech-notes/1600/1601.shtml>.


- 2** Enter the following MATLAB command at the command line:

```
deploytool
```

The MATLAB product opens the Deployment Tool dialog box.



**3** Create a new project by clicking the New Project button  in the toolbar.


**4** Add files that you want to encapsulate by dragging them to the Deployment Tool, or by selecting them and clicking the Add Files button  in the toolbar.

**5** Set properties for building and packaging.


When you build you can create a debug version of your compiled models and can specify verbose output. The debug option lets you trace back to the point where you can identify if the failure occurred in the initialization of MCR, the function call, or the termination routine.

Click **Settings** to view and specify these and other settings or click the Settings button in the toolbar.

**6** Add classes (optional).

**7** Save the project by clicking the Save button  in the toolbar.

**8** Build the component.

Click the Build button  in the toolbar to start the build process.

The MATLAB Builder EX product copies intermediate source files to *project\_directory\src* and output files necessary for deployment (a DLL and a VBA file (.bas) to *project\_directory\distrib*). It also creates a component assembly containing the wrapper class and a component CTF file in the *\distrib* subdirectory of your project directory.

The .ctf file (component technology file) is required to support components that encapsulate MATLAB functions when running them on a user machine that does not have the MATLAB desktop installed.

The **Output** pane shows the output of the build process and informs you of any problems encountered. The resulting DLL is automatically registered on your system.

**9** Test, edit, and rebuild as necessary.

## Using the `mcc` Command to Build a Component

Instead of using the Deployment Tool, you can use the `mcc` command on the MATLAB command line to build MATLAB Builder EX components. The following sections provide some examples of using the `mcc` command. See the MATLAB® Compiler™ documentation for a complete description of this command and its options, and `mcc` in the builder documentation for information on using it specifically with the builder.

---

**Note** If you use `mcc`, the *project\_directory\src* and *project\_directory\distrib* directories are not automatically created. To create these directories and copy associated files to them, use the `mcc` command's `-d` option.

---

## Testing the Component

After you build a component, you can test your software by importing the VBA file (.bas) into the Microsoft® Excel® Visual Basic® Editor and invoking one of the functions from the Excel worksheet. To import the VBA code into the Excel Visual Basic Editor:



- 1 Open Excel and select **Tools > Macros > Visual Basic Editor**.
- 2 From the Visual Basic Editor, select **File > Import** and select the created VBA file from the <project\_dir>\distrib directory.

The Visual Basic module created when you build the project contains the necessary initialization code and a VBA formula function for each MATLAB function processed. Each supplied formula function wraps a call to the respective compiled function in a format that can be accessed from a cell in an Excel worksheet. The function takes a list of inputs corresponding to the inputs of the original MATLAB function and returns a single output corresponding to the first output argument.

Formula functions of this type are most useful to access a function of one or more inputs that returns a single scalar value. When you require multiple outputs or outputs representing ranges of data, you need a more general Visual Basic subroutine. For details about integrating MATLAB Builder EX components into Microsoft Excel via Visual Basic for Applications, see Chapter 2, “Programming with the MATLAB® Builder™ EX Product”.

## Deploying the Component

After you create and test your component, you then create an Excel add-in (.xla) from the VBA code generated by the MATLAB Builder EX product by saving the worksheet file as an .xla file to the <project\_dir>\distrib directory.

---

**Note** You must have administrator privileges to build and deploy Excel Add-ins.

---


For more information about creating an Excel Add-in, refer to the Excel documentation on creating a .xla file.

- 1 Start Excel.
- 2 Select **Tools > Macros > Visual Basic Editor**.
- 3 In the Microsoft Visual Basic window, select **File > Import**.

- 4** Select VBA file (.bas) from the <projectdir>distrib directory.
- 5** Close the Visual Basic Editor.
- 6** From the Excel worksheet window, select **File > Save As**.
- 7** Set **Save as** to Microsoft® Excel® add-in (\*.xla).
- 8** Save the .xla file to <projectdir>\distrib.

You can also deploy files in default Excel file format and \*.bas formats. To deploy in default Excel file format, follow the previous steps but change the **Save as** type in step 7 to the default Excel file format. To deploy as VBA code, follow steps 1 to 4 only.

## Packaging and Distributing the Component

After you have successfully compiled your models and created the Excel add-in, you can package the component for distribution to your end users by reopening the project in Deployment Tool and clicking the **Package** button  in the toolbar. Run the generated self-extracting executable on the target machine and repeat for each additional target machine.

The MATLAB Builder EX product creates a self-extracting executable containing the following files.

<b>File</b>	<b>Description</b>
<componentname>.ctf	Component Technology File archive; platform-dependent file that must correspond to the end user's platform
<componentname_projectversion>.dll	Compiled component
_install.bat	Script run by the self-extracting executable

File	Description
MCRInstaller.exe	Self-extracting MATLAB Component Runtime library utility that installs the MCR; platform-dependent file that must correspond to the end user's platform; you must install on the target machine once per release. To include the MCR, ensure you select the <b>include MCR</b> option in the project settings. See "Working with the MCR" for more information.
*.xla	<p>User-written component Excel add-ins found in the &lt;projectdir&gt;\distrib directory.</p> <hr/> <p><b>Note</b> It is not possible to package MATLAB add-ins, such as <code>mfunction.xla</code>, by simply copying the file to the <code>distrib</code> directory. To do so, rename the add-in to <code>component_name.xla</code>.</p> <hr/>

To use the Excel add-ins:

- 1 Start Excel.
- 2 Select **Tools > Add-Ins**.
- 3 Select the desired .xla file.



# Programming with the MATLAB® Builder™ EX Product

---

Overview of the Integration Process (p. 2-3)	Provides information on integrating the MATLAB® Builder™ EX components into Microsoft® Excel® using the VBA programming environment
When to Use a Formula Function or a Subroutine (p. 2-4)	Discusses the two basic procedure types: functions and subroutines
Initializing the MATLAB® Builder™ EX Libraries with Excel® (p. 2-5)	Describes initializing the supporting libraries with the current instance of Excel®
Creating an Instance of a Class (p. 2-7)	Discusses creating an instance of the class that contains a class method
Calling the Methods of a Class Instance (p. 2-10)	Describes calling a class method to access the compiled MATLAB® functions
Processing varargin and varargout Arguments (p. 2-12)	Describes adding varargin and varargout parameters to the argument list of a class method
Calling Compiled MATLAB® Functions from Microsoft® Excel® (p. 2-14)	How to call compiled MATLAB functions from within an Excel spreadsheet

Handling Errors During a Method  
Call (p. 2-17)

Describes the Microsoft® Visual  
Basic® exception handling capability

Modifying Flags (p. 2-18)

Describes array formatting and data  
conversion flags

## Overview of the Integration Process

Each MATLAB® Builder™ EX component is built as a COM object that you can access from Microsoft® Excel® through Microsoft® Visual Basic® for Applications (VBA). This topic provides general information on how to integrate the MATLAB Builder EX components into Excel® using the VBA programming environment. It assumes that you have a working knowledge of VBA and is not intended to discuss how to program in Visual Basic®. Refer to the VBA documentation provided with Excel for general programming information.

You can integrate the MATLAB Builder EX components into a VBA project by creating a simple code module with functions and/or subroutines that load the necessary components, call methods as needed, and process any errors. In general, you need to address the following items in any code written to use the MATLAB Builder EX components:

- “When to Use a Formula Function or a Subroutine” on page 2-4
- “Initializing the MATLAB® Builder™ EX Libraries with Excel®” on page 2-5
- “Creating an Instance of a Class” on page 2-7
- “Calling the Methods of a Class Instance” on page 2-10
- “Processing varargin and varargout Arguments” on page 2-12
- “Handling Errors During a Method Call” on page 2-17
- “Modifying Flags” on page 2-18

---

**Note** All code samples in these topics are for illustration purposes and reference a hypothetical class named `myclass` contained in a component named `mycomponent` with a version number of 1.0.

---

## **When to Use a Formula Function or a Subroutine**

VBA provides two basic procedure types: functions and subroutines.

You access a VBA function directly from a cell in a worksheet as a formula function. Use function procedures when the original MATLAB® function takes one or more inputs and returns one scalar output.

You access a subroutine as a general macro. Use a subroutine procedure when the original MATLAB function returns an array of values or multiple outputs because you need to map these outputs into multiple cells/ranges in the worksheet.

When you create a component, the MATLAB® Builder™ EX product produces a VBA module (.bas file). This file contains simple call wrappers, each implemented as a function procedure for each method of the class.



## Initializing the MATLAB® Builder™ EX Libraries with Excel®

Before you use any MATLAB® Builder™ EX component, initialize the supporting libraries with the current instance of Excel®. Do this once for an Excel session that uses the MATLAB Builder EX components.

To do this initialization, call the utility library function `MWInitApplication`, which is a member of the `MWUtil` class. This class is part of the `MWComUtil` library. See “Utility Library Classes” on page B-3 for a detailed discussion of the functionality provided with this library.

One way to add this initialization code into a VBA module is to provide a subroutine that does the initialization once, and simply exits for all subsequent calls. The following Microsoft® Visual Basic® code sample initializes the libraries 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 reinitializing.

```
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
```

This code is similar to the default initialization code generated in the VBA module created when the component is built. Each function that uses the MATLAB Builder EX components can include a call to `InitModule` at the beginning to ensure that the initialization always gets performed as needed.

## Creating an Instance of a Class

### In this section...

“Overview” on page 2-7

“CreateObject Function” on page 2-7

“New Operator” on page 2-8

“How the MCR Is Shared Among Classes” on page 2-9

### Overview

Before calling a class method (compiled MATLAB® function), you must create an instance of the class that contains the method. VBA provides two techniques for doing this:

- CreateObject function
- New operator

### CreateObject Function

This method uses the Microsoft® Visual Basic® application program interface (API) CreateObject function to create an instance of the class. To use this method, Dim a variable of type Object to hold a reference to the class instance and call CreateObject using the class programmatic identifier (ProgID) as an argument, as shown in the next example:

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

    On Error Goto Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
    ' (call some methods on aClass)
    Exit Function
Handle_Error:
    foo = Err.Description
End Function
```

## New Operator

This method uses the Visual Basic® New operator on a variable explicitly dimensioned as the class to be created. Before using this method, you must reference the type library containing the class in the current VBA project. Do this by selecting the **Tools** menu from the Visual Basic Editor, and then selecting **References** to display the **Available References** list. From this list, select the necessary type library.

The following example illustrates using the New operator to create a class instance. It assumes that you have selected **mycomponent 1.0 Type Library** from the **Available References** list before calling this function.

```
Function foo(x1 As Variant, x2 As Variant) As Variant
    Dim aClass As mycomponent.myclass

    On Error Goto Handle_Error
    Set aClass = New mycomponent.myclass
    ' (call some methods on aClass)
    Exit Function
Handle_Error:
    foo = Err.Description
End Function
```

In this example, the class instance could be dimensioned as simply myclass. The full declaration in the form <component-name>.<class-name> guards against name collisions that could occur if other libraries in the current project contain types named myclass.

Both methods are equivalent in functionality. The first method does not require a reference to the type library in the VBA project, while the second results in faster code execution. The second method has the added advantage of enabling the **Auto-List-Members** and **Auto-Quick-Info** capabilities of the VBA editor to work with your classes. The default function wrappers created with each built component all use the first method for object creation.

In the previous two examples, the class instance used to make the method call was a local variable of the procedure. This creates and destroys a new class instance for each call. An alternative approach is to declare one single module-scoped class instance that is reused by all function calls, as in the initialization code of the previous example.

The following example illustrates this technique with the second method:

```
Dim aClass As mycomponent.myclass

Function foo(x1 As Variant, x2 As Variant) As Variant
    On Error Goto Handle_Error
    If aClass Is Nothing Then
        Set aClass = New mycomponent.myclass
    End If
    ' (call some methods on aClass)
    Exit Function
Handle_Error:
    foo = Err.Description
End Function
```

## How the MCR Is Shared Among Classes

The MATLAB® Builder™ EX product creates a single MATLAB Component Runtime (MCR) when the first Microsoft® COM class is instantiated in an application. This MCR is reused and shared among all subsequent class instances within the component, resulting in more efficient memory usage and eliminating the MCR startup cost in each subsequent class instantiation.

All class instances share a single MATLAB workspace and share global variables in the M-files used to build the component. This makes properties of a COM class behave as static properties instead of instance-wise properties.

## Calling the Methods of a Class Instance

After you have created a class instance, you can call the class methods to access the compiled MATLAB® functions. The MATLAB® Builder™ EX product applies a standard mapping from the original MATLAB function syntax to the method's argument list. See Chapter 6, "Utility Library for Microsoft® COM Components" for a detailed description of the mapping from MATLAB functions to COM class method calls.

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 compiled function and specifies how many outputs are requested. Methods that do not have output arguments do not pass a `nargout` argument. Following `nargout` are the output parameters listed in the same order as they appear on the left side of the original MATLAB function. Next come the input parameters listed in the same order as they appear on the right side of the original MATLAB function. All input and output arguments are typed as `Variant`, the default Visual Basic® data type.

The `Variant` type can hold any of the basic VBA types, arrays of any type, and object references. See "Data Conversion Rules" on page A-2 for a detailed description of how to convert `Variant` types 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 UDTs. You can also pass Microsoft® Excel® Range objects directly as input and output arguments.

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.

The following examples illustrate the process of passing input and output parameters from VBA to the MATLAB Builder EX component class methods.

The first example is a formula function that takes two inputs and returns one output. This function dispatches the 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
    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 same function as a subroutine and uses Excel ranges for input and output.

```
Sub foo(Rout As Range, Rin1 As Range, Rin2 As Range)
    Dim aClass As Object

    On Error Goto Handle_Error
    aClass = CreateObject("mycomponent.myclass.1_0")
    Call aClass.foo(1,Rout,Rin1,Rin2)
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

## Processing varargin and varargout Arguments

When varargin and/or varargout are present in the MATLAB® function that you are using for the Excel® component, these parameters are added to the argument list of the class method as the last input/output parameters in the list. 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.

The following example creates a varargin array to call a method resulting from a MATLAB function of the form `y = foo(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)  
    foo = y  
    Exit Function  
Handle_Error:  
    foo = Err.Description  
End Function
```

The MWUtil class included in the MWComUtil utility library provides the MWPack helper function to create varargin parameters. See “Utility Library Classes” on page B-3 for more details.

The next example processes a varargout parameter into three separate Excel Ranges. This function uses the MWUnpack function in the utility library. The MATLAB function used is `varargout = foo(x1,x2)`.



```
Sub foo(Rout1 As Range, Rout2 As Range, Rout3 As Range, _
        Rin1 As Range, Rin2 As Range)
    Dim aClass As Object
    Dim aUtil As Object
    Dim v As Variant

    On Error Goto Handle_Error
    aUtil = CreateObject("MComUtil.MWUtil")
    aClass = CreateObject("mycomponent.myclass.1_0")
    Call aClass.foo(3,v,Rin1,Rin2)
    Call aUtil.MWUnpack(v,0,True,Rout1,Rout2,Rout3)
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

## Calling Compiled MATLAB® Functions from Microsoft® Excel®

In order to call compiled MATLAB® functions from within a Microsoft® Excel® spreadsheet, perform the following from the Development and Deployment machines, as specified.

---

**Note** in order for a function to be called using the Microsoft Excel function syntax (=myfunction(input)), the MATLAB function must return a single scalar output argument.

---

Perform the following steps on the Development machine:

- 1 Copy the following files to a work directory on your computer from the Examples directory (*matlabroot\toolbox\matlabxl\Examples*):
  - doubleit.m
  - incrementit.m
  - powerit.m
- 2 From the MATLAB Command Prompt, enter `mbuild -setup` and select a Visual C++® compiler.

---

**Note** This procedure was tested using Microsoft® Visual C++ 8.0.

---

- 3 Start the Deployment Tool by entering `deploytool` at the MATLAB Command Prompt.
- 4 In the Deployment Tool, select **File > New Deployment Project**. The New Deployment Project window appears.
- 5 Select **MATLAB Builder EX** from the left pane and **Excel Add-in** from the right pane. Enter `myexcefunctions` as the Project Name and click **OK**. A class named `myexcefunctionsclass` is created and appears in the right pane of the Deployment Tool GUI.

- 6** Select **Project > Add File** and select the file `doubleit.m` from your work directory. Repeat this step, adding the files `incrementit.m` and `powerit.m`.
- 7** Select **Tools > Build**. The following files will be generated:
  - `work_directory\myexcelfunctions\distrib\myexcelfunctions.bas`
  - `work_directory\myexcelfunctions\distrib\myexcelfunctions.ctf`
  - `work_directory\myexcelfunctions\distrib\myexcelfuncgtions_1_0.dll`
- 8** Package your component by doing the following:
  - a** If you have not yet installed the MATLAB Component Runtime (MCR) on your deployment machine(s), you will need to include the MCR in your deployment package. Do this by performing the following steps:
    - i** In the Deployment Tool, click **Settings**.
    - ii** Select **Packaging** in the left pane of the Deployment Project Settings window.
    - iii** Check **Include MATLAB Component Runtime (MCR)** in Packaging Settings.
    - iv** Click **OK**.
  - b** In the Deployment Tool, select **Tools > Package**.  
`work_directory\myexcelfunctions\distrib\myexcelfunctions_pkg.exe` will be generated.

---

**Note** You must have administrator privileges to build and deploy.

---

Perform the following steps on the Deployment machine:

- 1** Copy `myexcelfunctions_pkg.exe` to the deployment machine(s). Copy the file to a standard place for use with Microsoft Excel, such as `Office_Installation_Directory\Library\MATLAB` where `Office_Installation_Directory` is a directory such as `C:\Program Files\Microsoft Office\OFFICE11`.

- 2 Run `myexcelfunctions_pkg.exe` to extract the archive and register `myexcelfunctions_1_0.dll`. If you have also included `MCRInstaller.exe`, follow the prompts to install the MATLAB Component Runtime.
- 3 Start Microsoft Excel. The spreadsheet `Book1` should be open by default.
- 4 In Excel®, select **Tools > Macro > Visual Basic Editor**. The Microsoft® Visual Basic® Editor starts.
- 5 In the Microsoft Visual Basic Editor, select **File > Import File**.
- 6 Browse to `myexcelfunctions.bas`, which was extracted from `myexcelfunctions_pkg.exe` and click **Open**. In the Project Explorer, **Module1** appears under the **Modules** node beneath **VBAProject (Book1)**.
- 7 In the Microsoft Visual Basic Editor, select **View > Microsoft Excel**. You can now use the `doubleit`, `incrementit`, and `powerit` functions in your **Book1** spreadsheet.
- 8 Test the functions, by doing the following:
  - a Enter `=doubleit(2.5)` in cell **A1**.
  - b Enter `=incrementit(11,17)` in cell **A2**.
  - c Enter `=powerit(7,2)` in cell **A3**.  
You should see values **5**, **28**, and **43** in cells **A1**, **A2**, and **A3** respectively.
- 9 To use the `doubleit`, `powerit`, and `incrementit` functions in all your new Microsoft Excel spreadsheets, do the following:
  - a Select **File > Save As**.
  - b Change the **Save as type** option to **.xlt (Template)**.
  - c Browse to the `Office_Installation_Directory\XLSTART` directory.
  - d Save the file as `Office_Installation_Directory\XLSTART\Book.xlt`.

---

**Note** Your Microsoft Excel Macro Security level must be set at **Medium** or **Low** to save this template.

---

## Handling Errors During a Method Call

Errors that occur while creating a class instance or during a class method create an exception in the current procedure. 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 are handled this way, including errors within the original MATLAB® code. An exception creates a Visual Basic® `ErrObject` object in the current context in a variable called `Err`. (See the Visual Basic for Applications documentation for a detailed discussion on VBA error handling.) All of the examples in this section illustrate the typical error trapping logic used in function call wrappers for MATLAB® Builder™ EX components.

## Modifying Flags

### In this section...

“Overview” on page 2-18

“Array Formatting Flags” on page 2-18

“Data Conversion Flags” on page 2-21

### Overview

Each MATLAB® Builder™ EX component exposes a single read/write property named `MWFlags` of type `MWFlags`. The `MWFlags` property consists of two sets of constants: array formatting flags and data conversion flags. *Array formatting flags* affect the transformation of arrays, whereas *data conversion flags* deal with type conversions of individual array elements.

The data conversion flags change selected behaviors of the data conversion process from `Variants` to MATLAB® types and vice versa. By default, the MATLAB Builder EX components allow setting data conversion flags at the class level through the `MWFlags` class property. This holds true for all Visual Basic® types, with the exception of the MATLAB Builder EX `MWStruct`, `MWField`, `MWComplex`, `MWSparse`, and `MWArg` types. Each of these types exposes its own `MWFlags` property and ignores the properties of the class whose method is being called. The `MWArg` class is supplied specifically for the case when a particular argument needs different settings from the default class properties.

This section provides a general discussion of how to set these flags and what they do. See “Class `MWFlags`” on page B-10 for a detailed discussion of the `MWFlags` type, as well as additional code samples.

### Array Formatting Flags

Array formatting flags guide the data conversion to produce either a MATLAB cell array or matrix from general `Variant` data on input or to produce an array of `Variants` or a single `Variant` containing an array of a basic type on output.

The following examples assume that you have referenced the `MWComUtil` library in the current project by selecting **Tools > References** and selecting **MWComUtil 7.5 Type Library** from the list:

```
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
```

In addition, these examples assume you have referenced the COM object created with Builder EX (mycomponent) as mentioned in “New Operator” on page 2-8.

Here, two Variant variables, var1 and var2 are constructed with the same numerical data, but internally they are structured differently: var1 is a 2-by-2 array of Variants with each element containing a 1-by-1 Double, while var2 is a 1-by-1 Variant containing a 2-by-2 array of Doubles.

In the MATLAB Builder EX product, when using the default settings, both of these arrays will be converted to 2-by-2 arrays of doubles. This does not follow the general convention listed in COM VARIANT to the MATLAB Conversion Rules. According to these rules, var1 converts to a 2-by-2 cell array with each cell occupied by a 1-by-1 double, and var2 converts directly to a 2-by-2 double matrix.

The two arrays both convert to double matrices because the default value for the `InputArrayFormat` flag is `mwArrayFormatMatrix`. The `InputArrayFormat` flag controls how arrays of these two types are handled. This default is used because array data originating from Excel® ranges is always in the form of an array of Variants (like `var1` of the previous example), and the MATLAB functions most often deal with matrix arguments.

But what if you want a cell array? In this case, you set the `InputArrayFormat` flag to `mwArrayFormatCell`. Do this by adding the following line after creating the class and before the method call:

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

Setting this flag presents all array input to the compiled MATLAB function as cell arrays.

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.

`AutoResizeOutput` is used for Excel Range objects passed directly as output parameters. 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.

The `TransposeOutput` flag transposes all array output. This flag is useful when dealing with MATLAB functions that output one-dimensional arrays. By default, the MATLAB product realizes one-dimensional arrays as 1-by-n matrices (row vectors) that become rows in an Excel worksheet.

You may prefer worksheet columns from row vector output. This example auto-resizes and transposes an output range:

```
Sub foo(Rout As Range, Rin As Range )  
    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.foo(1,Rout,Rin)
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub

```

## Data Conversion Flags

Data conversion flags deal with type conversions of individual array elements. The two data conversion flags, `CoerceNumericToType` and `InputDateFormat`, govern how numeric and date types are converted from VBA to the MATLAB product. Consider the example:

```

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

```

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

If the original MATLAB function expects doubles 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. In such a case set the `CoerceNumericToType` flag to `mwTypeDouble`, causing the data converter to convert all numeric input to `double`. In the previous example, place the following line after creating the class and before calling the methods:

```

aClass.MWFlags.DataConversionFlags.CoerceNumericToType =
mwTypeDouble

```

The InputDateFormat flag controls how the VBA Date type is converted. This 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 y1 = New MWArg
    y1.MWFlags.DataConversionFlags.OutputAsDate = True
    Call aClass.foo(2, ytemp, y2, today)
    y1 = ytemp.Value
    Exit Sub
Handle_Error:
```

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



# Usage Examples

---

Magic Square Examples (p. 3-2)	Creates a magic square from a single input integer
Multiple Files and Variable Arguments Example (p. 3-6)	Plots a line from 1 to an input number
Spectral Analysis Example (p. 3-13)	Creates a comprehensive Excel® add-in to perform spectral analysis

---

**Note** You can also find usage examples on MATLAB Central. Set the Search field to File Exchange and search for one or more of the following:

- `InterpExcelDemo`
- `MatrixMathExcelDemo`
- `ExcelCurveFit`

---

**Note** You must have administrator privileges to build and deploy Excel Add-ins.

---

## Magic Square Examples

### In this section...

“About the Magic Square Examples” on page 3-2

“Creating the Project” on page 3-3

“Adding the MATLAB® Builder™ EX COM Function to Microsoft® Excel®” on page 3-3

“Illustration 1. Output Magic Square Results to Microsoft® Excel®” on page 3-4

“Illustration 2. Transpose the Output” on page 3-4

“Illustration 3. Resize the Output” on page 3-5

“Inspecting the Microsoft® Visual Basic® Code” on page 3-5

### About the Magic Square Examples

The M-file `mymagic` takes a single input, an integer, and creates a magic square of that size.

The Microsoft® Excel® file `mymagic.xls` uses this function in three different ways:

- The first illustration calls the function `mymagic` with a value of 4. The function returns a magic square of size 4 and populates a range of Excel® cells with that magic square.
- The second illustration uses the transpose flag to transpose a magic square of size 4.
- The third illustration resizes the output to a higher value and moves its location within the Excel worksheet.

---

**Note** To get started, copy the distributed directory `xlmagic` from `matlabroot\toolbox\matlabxl\examples\xlmagic` to `myfiles\work`.

---

## Creating the Project


- 1 From the MATLAB® command prompt, change directories to `myfiles\work`.
- 2 If you have not already done so, execute the following command in the MATLAB product:

```
mbuild -setup
```

Be sure to choose a supported compiler. See Supported Compilers.

- 3 Enter the `deploytool` command to open the Deployment Tool.
- 4 Create a project with the following settings.

Setting	Value
Project name	xlmagic
Class name	xlmagicclass
Project directory	The name of your work directory followed by the component name.
Show verbose output	Selected

- 5 Locate your work directory and navigate to the `myfiles\work\xlmagic` directory and add the `mymagic.m` file to the project.
- 6 Build the component by clicking the Build button  in the Deployment Tool toolbar.

The build process begins, and a log of the build appears in the Output pane of the Deployment Tool dialog box. The files that are needed for the component are copied to two newly created directories, `src` and `distrib`, in the `xlmagic` directory. A copy of the build log is placed in the `src` directory.

## Adding the MATLAB® Builder™ EX COM Function to Microsoft® Excel®

- 1 Start Microsoft Excel on your system.

**2** Open the file myfiles\work\xlmagic\mymagic.xls.

---

**Note** If an Excel prompt says that this file contains macros, click **Enable Macros** to run this example.

---

### Illustration 1. Output Magic Square Results to Microsoft® Excel®

From the Excel main window (not the Microsoft® Visual Basic® editor), open the Macro dialog box by pressing the **Alt** and **F8** keys simultaneously, or by clicking **Tools > Macro > Macros**.

Select mymagic from the list and click **Run**. This procedure returns a magic square of size 4 beginning in cell B2.

	A	B	C	D	E	F
1						
2		4	16	2	3	13
3			5	11	10	8
4			9	7	6	12
5			4	14	15	1
6						
7	The above example runs the macro "mymagic" which					
8	populates the cells B2 through E5 with a magic square of 4					
9	Select Tools->Macro-> Macros to run this example					

### Illustration 2. Transpose the Output

Reopen the Macro dialog box, select the mymagic\_transpose macro and click **Run**. This procedure returns a magic square of size 4 transposed, beginning in cell B14.

13						
14		4	16	5	9	4
15			2	11	7	14
16			3	10	6	15
17			13	8	12	1
18						
19	The above example runs the macro "mymagic_transpose" which					
20	transposes the results of a magic square of 4 and populates the					
21	cells B14 through E17					
22	Select Tools->Macro-> Macros to run this example					



### Illustration 3. Resize the Output

Reopen the Macro dialog box, select the mymagic\_resize macro, and click **Run**. This procedure returns a magic square of size 4 beginning in cell B32.

Change the value of 4 in cell A32 to a higher value and rerun this macro. A magic square of the size you specified in cell A32 is returned, beginning in cell B32.

27	The below example runs the macro "mymagic_resize" which								
28	has an initial range for a magic square of 4 but will resize if								
29	the output is larger. Gradually increase the number in cell A32 and rerun the macro.								
30	CAUTION: Resizing will over write any existing data in the target cells								
31									
32	8	64	2	3	61	60	6	7	57
33		9	55	54	12	13	51	50	16
34		17	47	46	20	21	43	42	24
35		40	26	27	37	36	30	31	33
36		32	34	35	29	28	38	39	25
37		41	23	22	44	45	19	18	48
38		49	15	14	52	53	11	10	56
39		8	58	59	5	4	62	63	1
40									

### Inspecting the Microsoft® Visual Basic® Code

- 1 From the Excel main window, click **Tools > Macro > Visual Basic Editor**.
- 2 When the Visual Basic® Editor opens, in the **Project - VBAProject** window, double-click to expand VBAProject (mymagic.xls)
- 3 Expand the Modules folder and double-click the Module1 module.

This opens the VB Code window with the code for this project.

## Multiple Files and Variable Arguments Example

In this section...
“Overview” on page 3-6
“Creating the Project” on page 3-6
“Adding the MATLAB® Builder™ EX COM Function to Microsoft® Excel®” on page 3-7
“Illustration 4: Calling myplot” on page 3-9
“Illustration 5: Calling mysum Four Different Ways” on page 3-10
“Illustration 6: myprimes Macro” on page 3-11
“Inspecting the Microsoft® Visual Basic® Code” on page 3-12

### Overview

The M-file, `myplot`, takes a single integer input and plots a line from 1 to that number.

The M-file, `mysum`, takes an input of `varargin` of type integer, adds all the numbers, and returns the result.

The M-file, `myprimes`, takes a single integer input `n` and returns all the prime numbers less than or equal to `n`.

The Microsoft® Excel® file, `xlmulti.xls`, demonstrates these functions in several ways.

---

**Note** To get started, copy the distributed directory `xlmulti` from `matlabroot\toolbox\matlabxl\examples\xlmulti` to `myfiles\work`.

---

### Creating the Project

**1** From the MATLAB® command prompt, change directories to `myfiles\work`.

- 2 If you have not already done so, execute the following command in the MATLAB product:

```
mbuild -setup
```


Be sure to choose a supported compiler. See Supported Compilers.

- 3 While in the MATLAB product, issue the following command to open Deployment Tool:

```
deploytool
```

- 4 Create a project with the following settings:

Setting	Value
Project name	xlmulti
Class name	xlmulticlass
Project directory	The name of your work directory followed by the component name. .
Show verbose output	Selected

- 5 Locate your work directory and navigate to the xlmulti directory, which contains the M-files for myplot, myprimes, and mysum functions. Add these files to the project.
- 6 Build the component by clicking the Build icon  in the Deployment Tool toolbar.

The build process begins, and a log of the build appears in the Output pane of the Deployment Tool dialog box. The files that are needed for the component are copied to two newly created directories, src and distrib, in the xlmulti directory. A copy of the build log is placed in the src directory.

## Adding the MATLAB® Builder™ EX COM Function to Microsoft® Excel®

- 1 Start Microsoft Excel on your system.

- 2 Open the file `myfiles\work\xlmulti\xlmulti.xls`.

---

**Note** If an Excel® prompt says that this file contains macros, click **Enable Macros** to run this example.

---

The example appears as shown:

	A	B	C	D	E	F	G	H	I	J	K
1											
2	<b>Sample: myplot</b>										
3											
4	In this simple example we are just plotting a line from 1 to whatever										
5	number is supplied as an argument to the function in cell A7.										
6											
7	0										
8											
9											
10	<b>Sample: mysum</b>										
11											
12	In the below example we are just adding up a series of numbers that										
13	are explicitly stated.										
14	55										
15											
16											
17	In the below example we are just adding up a series of numbers that										
18	are from a range of cells.										
19	55	1	2	3	4	5	6	7	8	9	10
20											
21											
22	In the below example, we are adding up 3 separate ranges of cells.										
23	The ranges do not need to be the same size nor do all the cells in the range need to have data in them.										
24	120	1	2	3	4	5	6	7	8	9	10
25		1	2	3	4	5	6		8	9	10
26		1	2	3							
27											
28											
29	In the below example, we are adding 10 to a range of cells.										
30	16	1	2	3							
31											
32											
33											
34	<b>Sample: myprimes</b>										
35											
36											
37	The below example runs the macro "myprimes" which										
38	has an initial range for 4 prime numbers but will resize if										
39	the output is larger. Gradually increase the number in cell A42 and rerun the macro.										
40	CAUTION: Resizing will over write any existing data in the target cells										
41											
42	10										

### Illustration 4: Calling myplot

This illustration calls the function `myplot` with a value of 4. To execute the function, make A7 (`=myplot(4)`) the active cell. Press **F2** and then **Enter**.

	A	B	C	D	E	F	G
1							
2	Sample: myplot						
3							
4	In this simple example we are just plotting a line from 1 to whatever						
5	number is supplied as an argument to the function in cell A7.						
6							
7	0						
8							

This procedure plots a line from 1 through 4 in a MATLAB Figure window. This graphic can be manipulated similarly to the way one would manipulate a figure in the MATLAB product. Some functionality, such as the ability to change line style or color, is not available.

The calling cell contains 0 because the function does not return a value.

### Illustration 5: Calling mysum Four Different Ways

This illustration calls the function mysum in four different ways:

- The first (cell A14) takes the values 1 through 10, adds them, and returns the result of 55 (=mysum(1,2,3,4,5,6,7,8,9,10)).
- The second (cell A19) takes a range object that is a range of cells with the values 1 through 10, adds them, and returns the result of 55 (=mysum(B19:K19)).
- The third (cell A24) takes several range objects, adds them, and returns the result of 120 (=mysum(B24:K24,B25:L25,B26:D26)). This illustration demonstrates that the ranges do not need to be the same size and that all the cells do not need a value.
- The fourth (cell A30) takes a combination of a range object and explicitly stated values, adds them, and returns the result of 16 (=mysum(10,B30:D30)).

10	Sample: mysum										
11											
12	In the below example we are just adding up a series of numbers that										
13	are explicitly stated.										
14	55										
15											
16											
17	In the below example we are just adding up a series of numbers that										
18	are from a range of cells.										
19	55	1	2	3	4	5	6	7	8	9	10
20											
21											
22	In the below example, we are adding up 3 separate ranges of cells.										
23	The ranges do not need to be the same size nor do all the cells in the range need to have data in them.										
24	120	1	2	3	4	5	6	7	8	9	10
25		1	2	3	4	5	6		8	9	10
26		1	2	3							
27											
28											
29	In the below example, we are adding 10 to a range of cells.										
30	16	1	2	3							
31											

This illustration runs when the Excel file is opened. To reactivate the illustration, activate the appropriate cell. Then press **F2** followed by **Enter**.

## Illustration 6: myprimes Macro

In this illustration, the macro `myprimes` calls the function `myprimes.m` with an initial value of 10 in cell A42. The function returns all the prime numbers less than 10 to cells B42 through E42.

34	Sample: myprimes			
35				
36				
37	The below example runs the macro "myprimes" which			
38	has an initial range for 4 prime numbers but will resize if			
39	the output is larger. Gradually increase the number in cell A42 and rerun the macro.			
40	CAUTION: Resizing will over write any existing data in the target cells			
41				
42	10			
43				

To execute the macro, from the main Excel window (not the Visual Basic® Editor), open the Macro dialog box, by pressing the **Alt** and **F8** keys simultaneously, or by clicking **Tools > Macro > Macros**.

Select `myprimes` from the list and click **Run**.

34	Sample: myprimes							
35								
36								
37	The below example runs the macro "myprimes" which							
38	has an initial range for 4 prime numbers but will resize if							
39	the output is larger. Gradually increase the number in cell A42 and rerun the macro.							
40	CAUTION: Resizing will over write any existing data in the target cells							
41								
42	10	2	3	6	7			
43								

This function automatically resizes if the returned output is larger than the output range specified. Change the value in cell A42 to a number larger than 10. Then rerun the macro. The output returns all prime numbers less than the number you entered in cell A42.

34	Sample: myprimes								
35									
36									
37	The below example runs the macro "myprimes" which								
38	has an initial range for 4 prime numbers but will resize if								
39	the output is larger. Gradually increase the number in cell A42 and rerun the macro.								
40	CAUTION: Resizing will over write any existing data in the target cells								
41									
42	20	2	3	6	7	11	13	17	19

## Inspecting the Microsoft® Visual Basic® Code

- 1 On the Microsoft Excel main window, click **Tools > Macro > Visual Basic Editor**.
- 2 On the Microsoft® Visual Basic®, in the Project - VBAProject window, double-click to expand VBAProject (xlmulti.xls)
- 3 Expand the Modules folder and double-click the Module1 module. This opens the VB Code window with the code for this project.



# Spectral Analysis Example

## In this section...

“Overview” on page 3-13

“Building the Component” on page 3-14

“Integrating the Component Using VBA” on page 3-15

“Testing the Add-In” on page 3-27

“Packaging and Distributing the Add-In” on page 3-30

## Overview

This example illustrates the creation of a comprehensive Excel® add-in to perform spectral analysis. It requires knowledge of Visual Basic® forms and controls, as well as Excel workbook events. See the VBA documentation for a complete discussion of these topics.

The example creates an Excel add-in that performs a fast Fourier transform (FFT) on an input data set located in a designated worksheet range. The function returns the FFT results, an array of frequency points, and the power spectral density of the input data. It places these results into ranges you indicate in the current worksheet. You can also optionally plot the power spectral density.

You develop the function so that you can invoke it from the Excel **Tools** menu and can select input and output ranges through a GUI.

Creating the add-in requires four basic steps:

- 1** Build a standalone COM component from the MATLAB® code.
- 2** Implement the necessary VBA code to collect input and dispatch the calls to your component.
- 3** Create the GUI.
- 4** Create an Excel add-in and package all necessary components for application deployment.

## Building the Component

Your component will have one class with two methods, `computefft` and `plotfft`. The `computefft` method computes the FFT and power spectral density of the input data and computes a vector of frequency points based on the length of the data entered and the sampling interval. The `plotfft` method performs the same operations as `computefft`, but also plots the input data and the power spectral density in a MATLAB Figure window. The MATLAB code for these two methods resides in two M-files, `computefft.m` and `plotfft.m`.

```
computefft.m:
function [fftdata, freq, powerspect] = computefft(data, interval)
    if (isempty(data))
        fftdata = [];
        freq = [];
        powerspect = [];
        return;
    end
    if (interval <= 0)
        error('Sampling interval must be greater than zero');
        return;
    end
    fftdata = fft(data);
    freq = (0:length(fftdata)-1)/(length(fftdata)*interval);
    powerspect = abs(fftdata)/(sqrt(length(fftdata)));
plotfft.m:
function [fftdata, freq, powerspect] = plotfft(data, interval)
    [fftdata, freq, powerspect] = computefft(data, interval);
    len = length(fftdata);
    if (len <= 0)
        return;
    end
    t = 0:interval:(len-1)*interval;
    subplot(2,1,1), plot(t, data)
    xlabel('Time'), grid on
    title('Time domain signal')
    subplot(2,1,2), plot(freq(1:len/2), powerspect(1:len/2))
    xlabel('Frequency (Hz)'), grid on
    title('Power spectral density')
```

To proceed with the actual building of the component, follow these steps:

- 1 If you have not already done so, execute the following command in the MATLAB product:

```
mbuild -setup
```

Be sure to choose a supported compiler. See Supported Compilers.

- 2 Create a project with the following settings:

Setting	Value
Component name	Fourier
Class name	Fourier
Project directory	The name of your work directory followed by the component name.
Show verbose output	Selected

- 3 Add the `computefft.m` and `plotfft.m` M-files to the project.
- 4 Save the project. Make note of the project directory because you will refer to it later when you save your add-in.
- 5 Build the component by clicking the Build button in the Deployment Tool toolbar.

## Integrating the Component Using VBA

Having built your component, you can implement the necessary VBA code to integrate it into Excel.

### Selecting the Libraries

Follow these steps to open Excel and select the libraries you need to develop the add-in:

- 1 Start Excel on your system.

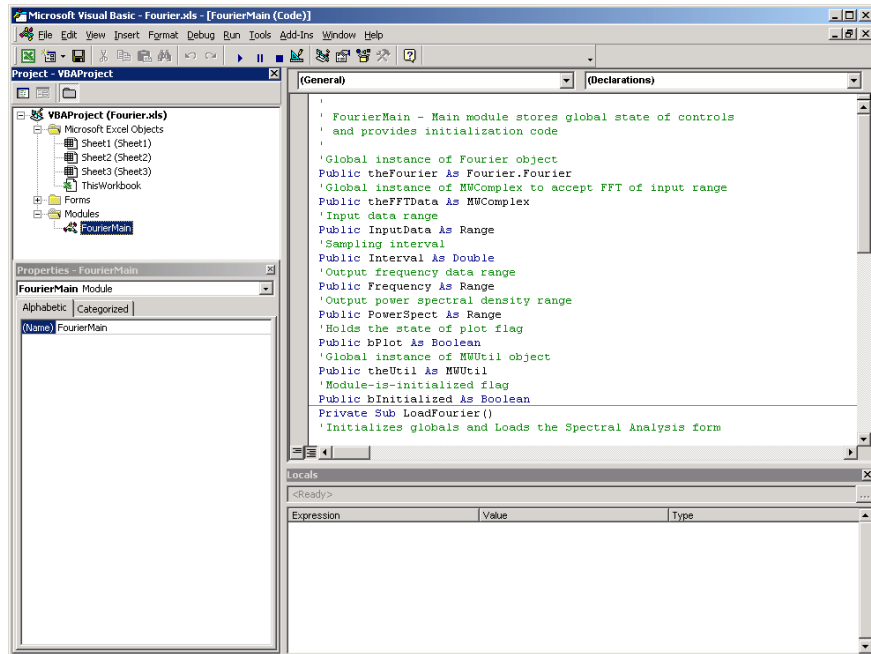
- 2 From the Excel main menu, click **Tools > Macro > Visual Basic Editor**.
- 3 When the Visual Basic Editor starts, click **Tools > References** to open the Project References dialog box.
- 4 Select **Fourier 1.0 Type Library** and **MWComUtil 7.5 Type Library** from the list.

**Creating the Main VB Code Module for the Application.** The add-in requires some initialization code and some global variables to hold the application's state between function invocations. To achieve this, implement a Visual Basic code module to manage these tasks, as follows:

- 1 Right-click the **VBAProject** item in the project window and click **Insert > Module**.

A new module appears under **Modules** in the **VBA Project**.

- 2 In the module's property page, set the Name property to **FourierMain**. See the next figure.



### 3 Enter the following code in the FourierMain module:

```

'
' FourierMain - Main module stores global state of controls
' and provides initialization code
'

Public theFourier As Fourier.Fourierclass 'Global instance of Fourier object
Public theFFTData As MWComplex 'Global instance of MWComplex to accept FFT
Public InputData As Range 'Input data range
Public Interval As Double 'Sampling interval
Public Frequency As Range 'Output frequency data range
Public PowerSpect As Range 'Output power spectral density range
Public bPlot As Boolean 'Holds the state of plot flag
Public theUtil As MWUtil 'Global instance of MWUtil object
Public bInitialized As Boolean 'Module-is-initialized flag

Private Sub LoadFourier()
'Initializes globals and Loads the Spectral Analysis form
Dim MainForm As frmFourier

```

```
        On Error GoTo Handle_Error
        Call InitApp
        Set MainForm = New frmFourier
        Call MainForm.Show
        Exit Sub
Handle_Error:
        MsgBox (Err.Description)
End Sub

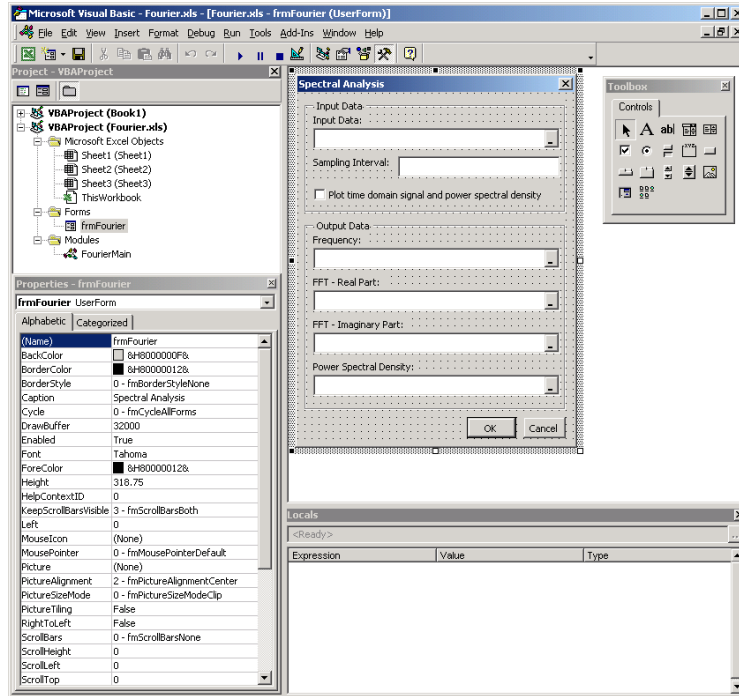
Private Sub InitApp()
'Initializes classes and libraries. Executes once
'for a given session of Excel
    If bInitialized Then Exit Sub
    On Error GoTo Handle_Error
    If theUtil Is Nothing Then
        Set theUtil = New MWUtil
        Call theUtil.MWInitApplication(Application)
    End If
    If theFourier Is Nothing Then
        Set theFourier = New Fourier.Fourierclass
    End If
    If theFFTData Is Nothing Then
        Set theFFTData = New MWComplex
    End If
    bInitialized = True
    Exit Sub
Handle_Error:
        MsgBox (Err.Description)
End Sub
```

### Creating the Visual Basic® Form

The next step in the integration process develops a user interface for your add-in using the Visual Basic Editor. Follow these steps to create a new user form and populate it with the necessary controls:

- 1 Right-click **VBAProject** in the VBA project window and click **Insert > UserForm**.

A new form appears under Forms in the VBA project window.



- 2 In the form's property page, set the Name property to frmFourier and the Caption property to Spectral Analysis.
- 3 Add a series of controls to the blank form to complete the dialog box, as summarized in the following table:

### Controls Needed for Spectral Analysis Example

Control Type	Control Name	Properties	Purpose
CheckBox	chkPlot	Caption = Plot time domain signal and power spectral density	Plots input data and power spectral density.

**Controls Needed for Spectral Analysis Example (Continued)**

<b>Control Type</b>	<b>Control Name</b>	<b>Properties</b>	<b>Purpose</b>
CommandButton	btnOK	Caption = OK Default = True	Executes the function and dismisses the dialog box.
CommandButton	btnCancel	Caption = Cancel Cancel = True	Dismisses the dialog box without executing the function.
Frame	Frame1	Caption = Input Data	Groups all input controls.
Frame	Frame2	Caption = Output Data	Groups all output controls.
Label	Label1	Caption = Input Data:	Labels the RefEdit for input data.
TextBox	edtSample	N/A	N/A
Label	Label2	Caption = Sampling Interval	Labels the TextBox for sampling interval.
Label	Label3	Caption = Frequency:	Labels the RefEdit for frequency output.
Label	Label4	Caption = FFT - Real Part:	Labels the RefEdit for real part of FFT.
Label	Label5	Caption = FFT - Imaginary Part:	Labels the RefEdit for imaginary part of FFT.



**Controls Needed for Spectral Analysis Example (Continued)**

<b>Control Type</b>	<b>Control Name</b>	<b>Properties</b>	<b>Purpose</b>
Label	Label6	Caption = Power Spectral Density	Labels the RefEdit for power spectral density.
RefEdit	refedtInput	N/A	Selects range for input data.
RefEdit	refedtFreq	N/A	Selects output range for frequency points.
RefEdit	refedtReal	N/A	Selects output range for real part of FFT of input data.
RefEdit	refedtImag	N/A	Selects output range for imaginary part of FFT of input data.
RefEdit	refedtPowSpect	N/A	Selects output range for power spectral density of input data.

The following figure shows the controls layout on the form:

- 4 When the form and controls are complete, right-click the form and click **View Code**.

The following code listing shows the code to implement. Notice that this code references the control and variable names listed in Controls Needed for Spectral Analysis Example on page 3-19. If you used different names for any of the controls or any global variable, change this code to reflect those differences.

```

'
'frmFourier Event handlers
'
Private Sub UserForm_Activate()
'UserForm Activate event handler. This function gets called before
'showing the form, and initializes all controls with values stored
'in global variables.
    On Error GoTo Handle_Error
    If theFourier Is Nothing Or theFFTDData Is Nothing Then Exit Sub
    'Initialize controls with current state
    If Not InputData Is Nothing Then
        refedInput.Text = InputData.Address
    End If

```

```

edtSample.Text = Format(Interval)
If Not Frequency Is Nothing Then
    refedtFreq.Text = Frequency.Address
End If
If Not IsEmpty (theFFTDData.Real) Then
If IsObject(theFFTDData.Real) And TypeOf theFFTDData.Real Is Range Then
    refedtReal.Text = theFFTDData.Real.Address
    End If
End If
If Not IsEmpty (theFFTDData.Imag) Then
If IsObject(theFFTDData.Imag) And TypeOf theFFTDData.Imag Is Range Then
    refedtImag.Text = theFFTDData.Imag.Address
    End If
End If
If Not PowerSpect Is Nothing Then
    refedtPowSpect.Text = PowerSpect.Address
End If
chkPlot.Value = bPlot
Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub btnCancel_Click()
'Cancel button click event handler. Exits form without computing fft
'or updating variables.
    Unload Me
End Sub
Private Sub btnOK_Click()
'OK button click event handler. Updates state of all variables from controls
'and executes the computefft or plotfft method.
    Dim R As Range

    If theFourier Is Nothing Or theFFTDData Is Nothing Then GoTo Exit_Form
    On Error Resume Next
    'Process inputs
    Set R = Range(refedtInput.Text)
    If Err <> 0 Then
        MsgBox ("Invalid range entered for Input Data")
    End If
    Exit Sub

```

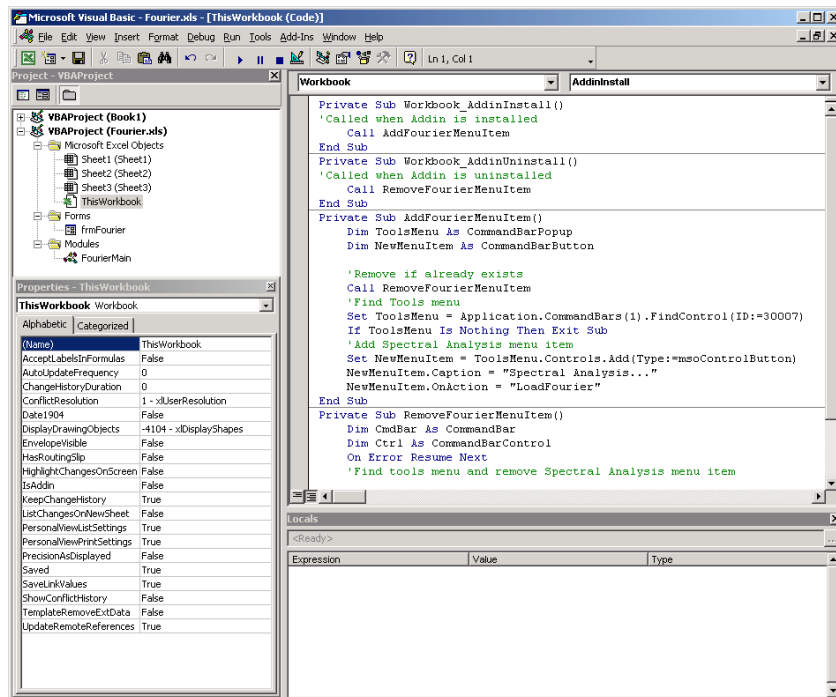
```
End If
Set InputData = R
Interval = CDBl(edtSample.Text)
If Err <> 0 Or Interval <= 0 Then
    MsgBox ("Sampling interval must be greater than zero")
    Exit Sub
End If
'Process Outputs
Set R = Range(refedtFreq.Text)
If Err = 0 Then
    Set Frequency = R
End If
Set R = Range(refedtReal.Text)
If Err = 0 Then
    theFFTData.Real = R
End If
Set R = Range(refedtImag.Text)
If Err = 0 Then
    theFFTData.Imag = R
End If
Set R = Range(refedtPowSpect.Text)
If Err = 0 Then
    Set PowerSpect = R
End If
bPlot = chkPlot.Value
'Compute the fft and optionally plot power spectral density
If bPlot Then
    Call theFourier.plotfft(3, theFFTData, Frequency, PowerSpect, _
        InputData, Interval)
Else
    Call theFourier.computefft(3, theFFTData, Frequency, PowerSpect, _
        InputData, Interval)
End If
GoTo Exit_Form
Handle_Error:
    MsgBox (Err.Description)
Exit_Form:
    Unload Me
End Sub
```

## Adding the Spectral Analysis Menu Item to Excel®

The last step in the integration process adds a menu item to Excel so that you can open the tool from the Excel **Tools** menu. To do this you add event handlers for the workbook's `AddinInstall` and `AddinUninstall` events that install and uninstall menu items. The menu item calls the `LoadFourier` function in the `FourierMain` module.

Follow these steps to implement the menu item:

- 1 Right-click the **ThisWorkbook** item in the VBA project window and click **View Code**.



- 2 Place the following code into `ThisWorkbook`.

```

Private Sub Workbook_AddinInstall()
    'Called when Addin is installed
    Call AddFourierMenuItem
  
```

```
End Sub

Private Sub Workbook_AddinUninstall()
'Called when Addin is uninstalled
    Call RemoveFourierMenuItem
End Sub

Private Sub AddFourierMenuItem()
    Dim ToolsMenu As CommandBarPopup
    Dim NewMenuItem As CommandBarButton

    'Remove if already exists
    Call RemoveFourierMenuItem
    'Find Tools menu
    Set ToolsMenu = Application.CommandBars(1).FindControl(ID:=30007)
    If ToolsMenu Is Nothing Then Exit Sub
    'Add Spectral Analysis menu item
    Set NewMenuItem = ToolsMenu.Controls.Add(Type:=msoControlButton)
    NewMenuItem.Caption = "Spectral Analysis..."
    NewMenuItem.OnAction = "LoadFourier"
End Sub

Private Sub RemoveFourierMenuItem()
    Dim CmdBar As CommandBar
    Dim Ctrl As CommandBarControl
    On Error Resume Next
    'Find tools menu and remove Spectral Analysis menu item
    Set CmdBar = Application.CommandBars(1)
    Set Ctrl = CmdBar.FindControl(ID:=30007)
    Call Ctrl.Controls("Spectral Analysis...").Delete
End Sub
```

### 3 Save the add-in.

Now that the VBA coding is complete, you can save the add-in. Save this file into the <project-directory>\distrib directory that Deployment Tool created when building the project. Here, <project-directory> refers to the project directory that Deployment Tool used to save the Fourier project. Name the add-in Spectral Analysis.

- a. From the Excel main menu, select **File > Properties**.
- b. When the Workbook Properties dialog box appears, click the **Summary** tab, and enter Spectral Analysis as the workbook title.
- c. Click **OK** to save the edits.
- d. From the Excel main menu, click **File > Save As**.
- e. When the Save As dialog box appears, select Microsoft Excel Add-In (\*.xla) as the file type, and browse to <project-directory>\distrib.
- f. Enter Fourier.xla as the file name and click **Save** to save the add-in.

## Testing the Add-In

Before distributing the add-in, test it with a sample problem.

Spectral analysis is commonly used to find the frequency components of a signal buried in a noisy time domain signal. In this example you will create a data representation of a signal containing two distinct components and add to it a random component. This data along with the output will be stored in columns of an Excel worksheet, and you will plot the time-domain signal along with the power spectral density.

## Creating the Test Problem

Follow these steps to create the test problem:

- 1** Start a new session of Excel with a blank workbook.
- 2** From the main menu click **Tools > Add-Ins**.
- 3** When the Add-Ins dialog box appears, click **Browse**.
- 4** Browse to the <project-directory>\distrib directory, select Fourier.xla, and click **OK**.

The **Spectral Analysis** add-in appears in the available **Add-Ins** list and is selected.

- 5** Click **OK** to load the add-in.

This add-in installs a menu item under the Excel **Tools** menu. You can display the Spectral Analysis GUI by selecting **Tools>Spectral Analysis**. Before invoking the add-in, create some data, in this case a signal with components at 15 and 40 Hz. Sample the signal for 10 seconds at a sampling rate of 0.01 s. Put the time points into column A and the signal points into column B.

### **Creating the Data**

Follow these steps to create the data:

- 1** Enter 0 for cell A1 in the current worksheet.
- 2** Click cell A2 and type the formula "`= A1 + 0.01`".
- 3** Click and hold the lower-right corner of cell A2 and drag the formula down the column to cell A1001. This procedure fills the range A1:A1001 with the interval 0 to 10 incremented by 0.01.
- 4** Click cell B1 and type the following formula

```
"= SIN(2*PI()*15*A1) + SIN(2*PI()*40*A1) + RAND() "
```

Repeat the drag procedure to copy this formula to all cells in the range B1:B1001.

### **Running the Test**

Using the column of data (column B), test the add-in as follows:

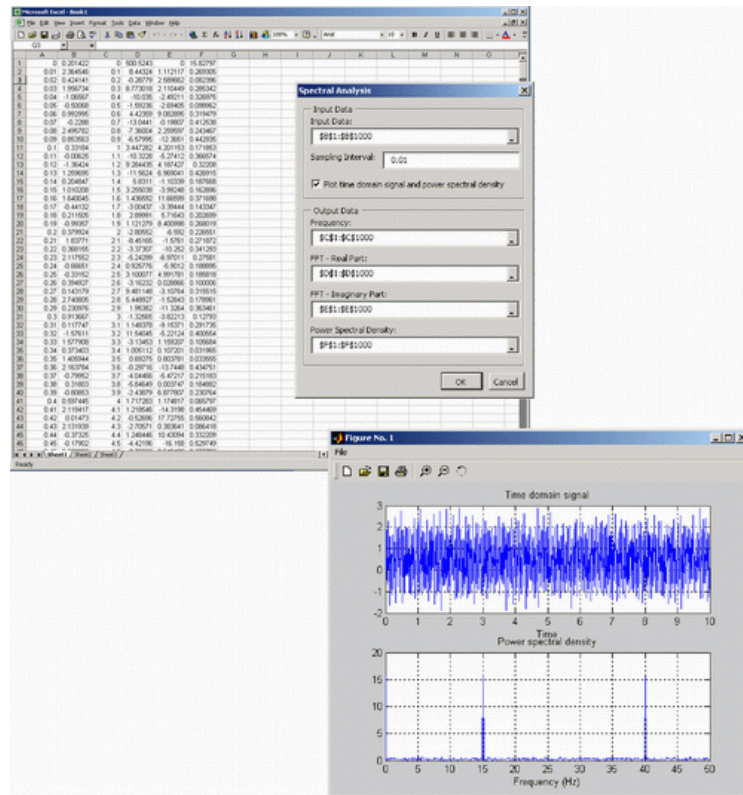
- 1** Select **Tools > Spectral Analysis** from the main menu.
- 2** Click the **Input Data** box.
- 3** Select the B1:B1001 range from the worksheet, or type this address into the **Input Data** field.
- 4** In the **Sampling Interval** field, type 0.01.
- 5** Select **Plot time domain signal and power spectral density**.



6 Enter C1:C1001 for frequency output, and likewise enter D1:D1001, E1:E1001, and F1:F1001 for the FFT real and imaginary parts, and spectral density.

7 Click **OK** to run the analysis.

The next figure shows the output.



The power spectral density reveals the two signals at 15 and 40 Hz.

## Packaging and Distributing the Add-In

As a final step, package the add-in, the COM component, and all supporting libraries into a self-extracting executable. This package can be installed onto other computers that need to use the Spectral Analysis add-in.

To package and distribute the add-in, follow these steps:

- 1** Reopen project in the Deployment Tool, if it is not already open.
- 2** Select **Include MCR** in the project settings, if the MCR is required on the deployment machine.
- 3** Click the Package button in the toolbar.

The MATLAB® Builder™ EX product creates the `Fourier_pkg.exe` self-extracting executable.

- 4** To install this add-in onto another computer, copy the `Fourier_pkg.exe` package to that machine, run it from a command prompt, and follow the instructions in the `readme.txt` file that is automatically generated with your packaged output.

# Function Wizard

---

Overview of the Function Wizard (p. 4-2)	Describes the purpose and use of the Function Wizard
Installing the Function Wizard Add-In (p. 4-3)	How to install the Add-In
Starting the Function Wizard (p. 4-5)	How to open the Function Viewer
Understanding the Function Viewer (p. 4-6)	How to load and execute functions
Component Browser (p. 4-8)	How to view components currently installed
Function Properties (p. 4-9)	How to edit inputs and outputs to functions
Argument Properties (p. 4-14)	How to select worksheet ranges and specify values
Function Utilities (p. 4-16)	How to rename, copy, and move functions

## Overview of the Function Wizard

The Function Wizard enables you to pass Microsoft® Excel® (Excel® 2000 or later) worksheet values to a compiled MATLAB® model and to return model output to a cell or range of cells in the worksheet. The Function Wizard provides an intuitive interface to Excel worksheets. Knowledge of Microsoft® Visual Basic® for Applications (VBA) programming is not required.

The Function Wizard reflects any changes that you make in the worksheets, such as range selections. Going in the opposite direction, you can use the Function Wizard to control the placement and output of data from MATLAB functions to the worksheets.

The Function Wizard does not currently support the MATLAB struct, sparse, and complex data types.

## Installing the Function Wizard Add-In

The Function Wizard GUI is contained in an Excel® add-in (`mlfunction.xla`) residing in the `matlabroot\toolbox\matlabxl\matlabxl` directory. You must install this add-in before using the Function Wizard.

To install the add-in:

The Function Wizard is not packaged by default with deployed components. To distribute the wizard, place `mlfunction.xla` in the top-level directory of the installed component, substituting the installed component's directory.

### Installing with Versions of Office Older Than 2007


Do the following to install the add-in with versions of Office prior to Office 2007:

- 1 Select **Tools > Add-Ins** from the Excel main menu.
- 2 If the Function Wizard was previously installed, a reference to **MATLAB® Function Wizard** appears in the list. Select the item and click **OK**.

If the Function Wizard was not previously installed, click **Browse** and proceed to the `matlabroot\toolbox\matlabxl\matlabxl` directory. Select `mlfunction.xla`. Click **OK** in this dialog box and in the preceding one.

### Installing with Office 2007

Do the following to install the add-in with Office 2007:

- 1 Click the Microsoft® Office Button .
- 2 Select **Excel Options**.
- 3 Select Add-ins. Under Manage, select Excel Add-ins and click Go.
- 4 Browse to `matlabroot/toolbox/matlabxl/matlabxl` and select the **MATLAB Function Wizard** by choosing `mlfunction.xla`.

---

**Note** The add-in may appear as **mfunction** if it has previously been installed.

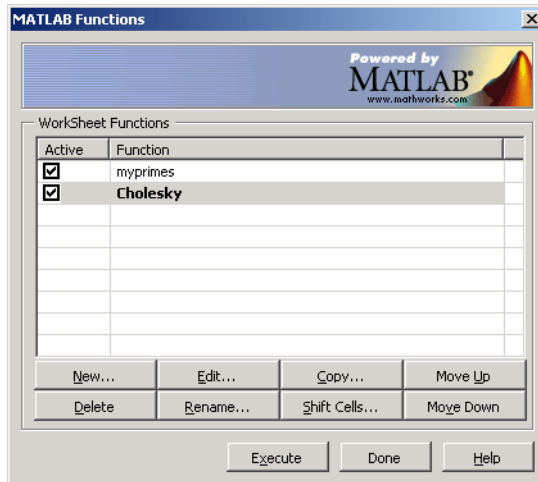
---

## Starting the Function Wizard

Start the Function Wizard in one of the following ways depending on what version of Office you have installed.

### Starting the Function Wizard with Versions of Microsoft® Office Older Than 2007

To start the Function Wizard, click **Tools > MATLAB Functions** from the Excel® menu bar. The starting point of the Function Wizard, called the Function Viewer, now appears:



### Starting the Function Wizard with Office 2007

On the toolbar, select **Add-Ins** and then select **MATLAB Functions**. The Function Wizard opens.

## Understanding the Function Viewer

In this section...
“Using the Function Viewer” on page 4-6
“Loading and Executing Functions” on page 4-6

The Function Viewer controls the execution of worksheet functions. Use the Function Viewer to organize the list of all currently loaded MATLAB® Builder™ EX functions.

### Using the Function Viewer

The Function Viewer displays the names of all loaded functions. You can edit this name to provide a more descriptive identifier. A check box for each entry denotes the active/inactive state of each function. Inactive functions are not executed when you click **Execute**.

Below the function list is a group of eight buttons. To add a new component to the list of loaded worksheet functions, click **New** (see “Component Browser” on page 4-8).

Each of the other buttons performs a specific action on the currently selected function. To select a function, left-click the list item. The row becomes selected. You can change the current selection by left-clicking a different list item, or by using the up and down arrow keys on your keyboard.

### Loading and Executing Functions

To load and execute an MATLAB Builder EX function in your worksheet requires three steps:

- 1 Load an MATLAB Builder EX component.

Click **New** on the Function Viewer to display the **Component Browser**. (See “Component Browser” on page 4-8.) Use this browser to select the component you want to load from the list of all currently installed MATLAB Builder EX components. From the selected component, add the method that you want to call.



- 2** Set the inputs, outputs, and other properties of your function.

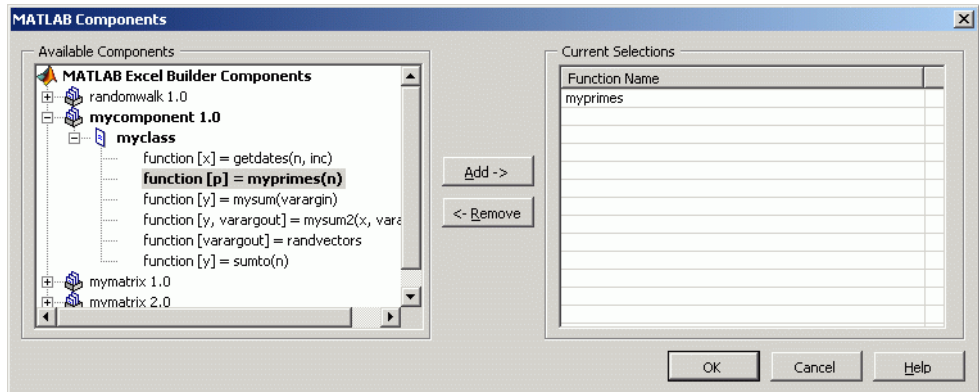
Click **Edit** to display the Function Properties dialog box. (See “Function Properties” on page 4-9.)

- 3** Click **Execute** on the Function Viewer.

When you click **Execute**, functions execute in the order displayed in the list.

## Component Browser

The Component Browser lists all MATLAB® Builder™ EX components currently installed on the system. When you click **New** on the Function Viewer, this dialog box appears:



The Component Browser lists each component by name and version. Expanding a component reveals the class name at the next level. You can also expand the class to reveal the MATLAB® functions that make up the class methods.

Select the desired method and click **Add** to add a function. To load all methods of a class, select the class name and click **Add**. Added functions appear under **Current Selections** on the right of the browser.

To remove a function before returning to the Function Viewer, select it under **Current Selections** and click **Remove**.

## Function Properties

### In this section...

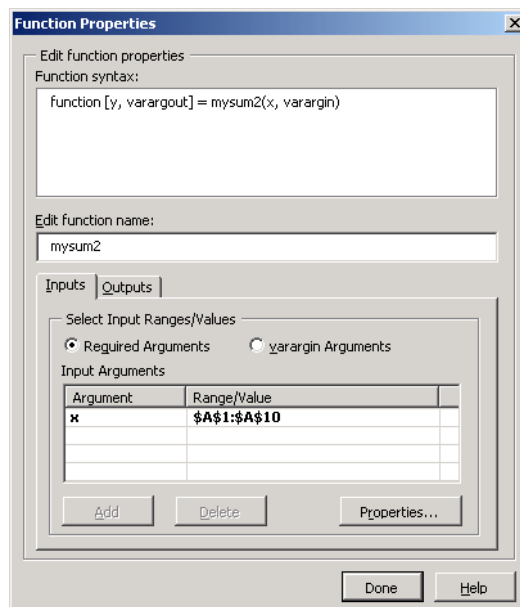
“The Function Properties Dialog Box” on page 4-9

“Editing Function Arguments” on page 4-10

### The Function Properties Dialog Box

This group of dialog boxes sets properties and values for the inputs and outputs. You can map inputs and outputs to ranges in your worksheet. You can also rename a function with any of these dialog boxes.

When you click **Edit** on the Function Viewer, the Function Properties dialog box appears, as shown:



The **Add** and **Delete** buttons become active when you click **varargin Arguments**.

Click the **Outputs** tab to switch to editing outputs.

### Editing Function Arguments

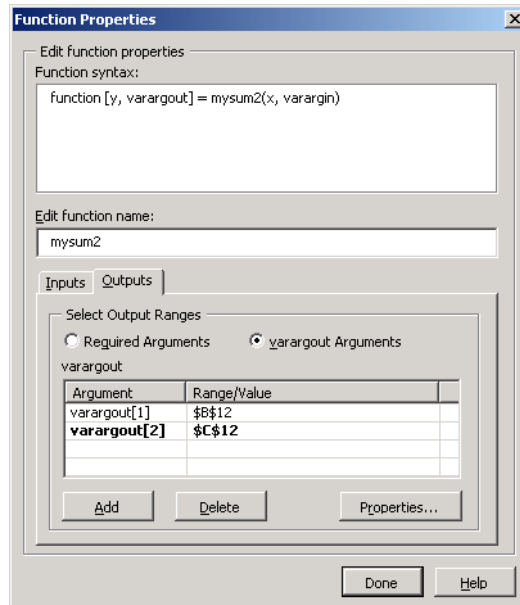
Function arguments may be either required arguments or varargin/varargout arguments:

- Required arguments appear first on the left or right sides of a MATLAB® function and are not named varargin or varargout.
- varargin/varargout arguments always appear as the last input or output. They let you specify a variable number of arguments.

### Editing Required and Varargin/Varargout Arguments

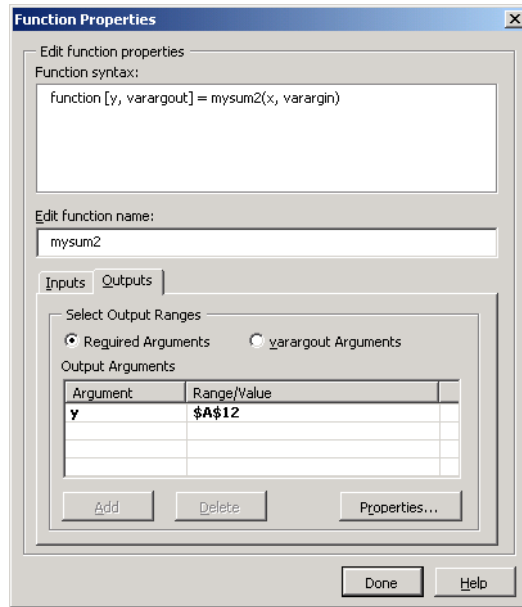
To edit required arguments, select the argument from the list and click **Properties**.

Before you can edit varargin/varargout arguments, you must first explicitly add them using **Add**. If the MATLAB function does not have varargin/varargout arguments, the ability to add arguments to the list is disabled. After you have added varargin/varargout arguments, you can edit them in the same way as required arguments. When you are editing varargin/varargout arguments, the Function Properties dialog box appears as shown:



### Editing Required Outputs

When you are editing required output arguments, the Function Properties dialog box appears as shown:

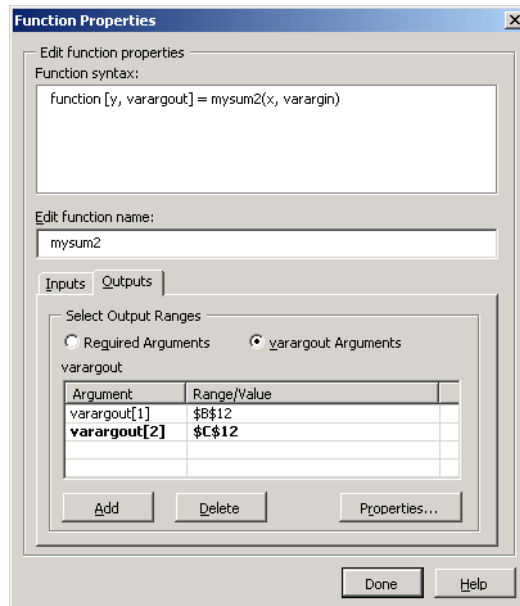


The **Add** and **Delete** buttons become active when you click **varargout Arguments**.

Click the **Inputs** tab to switch to editing inputs.

## Editing varargout Outputs

When you are editing varargout outputs, the Function Properties dialog box appears as shown:



## Argument Properties

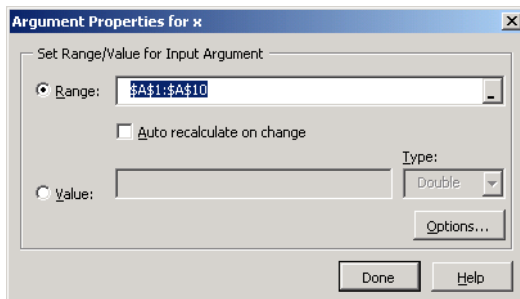
### In this section...

“Input Argument Properties Dialog Box” on page 4-14

“Output Argument Properties Dialog Box” on page 4-15

### Input Argument Properties Dialog Box

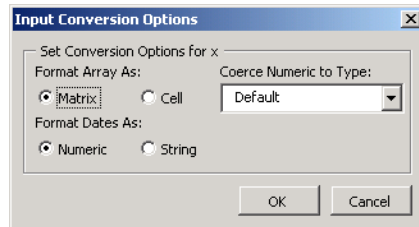
Here is an example of the Argument Properties dialog box for input arguments. In this example, the input arguments have a range of A1 to A10.



From this dialog box you can

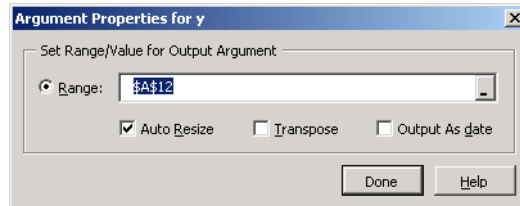
- Select the **Range** list to specify a range of current input arguments.
- Click **Auto recalculate on change** to tell the MATLAB® product to recalculate the current function when any cell in the current argument changes.
- Select the **Value** list to set a single value for the current argument. Then select the type from the **Type** list.
- Click **Options** to set the conversion options. Then set the options in the Input Conversion Options dialog box as shown:





## Output Argument Properties Dialog Box

Here is an example of the Argument Properties dialog box for output arguments. In this example, the output argument is A12.



From this dialog box you can

- From the **Range** list, select the worksheet range to be used as the output argument.
- Select **Auto resize** to tell the MATLAB product to adjust the output range to fit the output array. This setting is useful 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.
- Select **Transpose output** to transpose the output arguments. This setting is useful when calling a component where the MATLAB function returns outputs as row vectors, and you want the data in columns.
- Select **Output as date** to coerce the output values to become Excel dates.

## Function Utilities

### In this section...

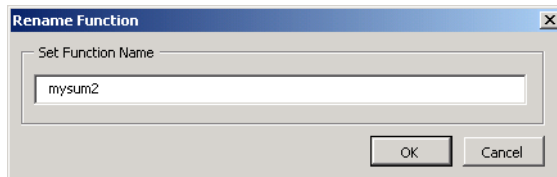
“Rename Function Dialog Box” on page 4-16

“Copy Function Dialog Box” on page 4-16

“Move Function Dialog Box” on page 4-17

### Rename Function Dialog Box

Use the Rename Function dialog box to rename a function. To open this dialog box, click **Rename** on the Function Viewer. Here is an example of this dialog box, with `mysum2` as the new function name:



In this dialog box, you can

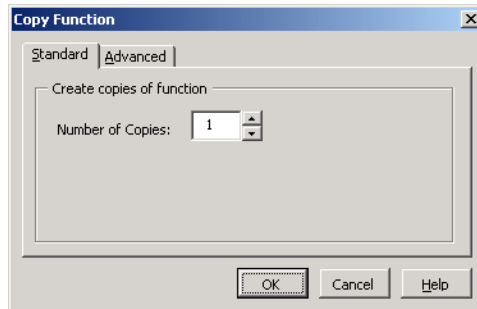
- Enter a new name for the selected function.
- Click **OK** to save the new name and return to the Function Viewer.
- Click **Cancel** to return to the Function Viewer without saving the new name.

### Copy Function Dialog Box

Use the Copy Function dialog box to make copies of the current function. To open this dialog box, click **Copy** on the Function Viewer.

The Copy Function dialog box has two tabs:

- The **Standard** tab creates a specified number of copies of the function while copying any argument/range values you have set. Here is an illustration of this dialog box, with the number of copies, set to 1:

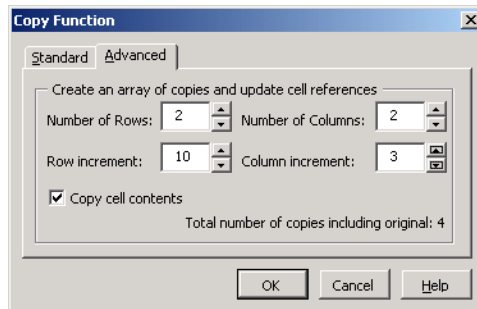


- The **Advanced** tab creates a rectangular array of copies of the current function in the current worksheet, and optionally copies the cell contents of ranges referenced by the function's arguments.

When you set the number of rows and columns and the row/column increments, the copy process automatically updates cell references by the specified increment amounts.

- Positive increments move rows down and columns to the right.
- Negative increments move rows up and columns to the left.

The following example shows the **Advanced** tab:



## Move Function Dialog Box

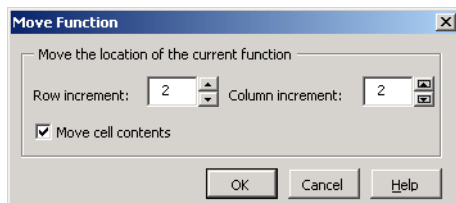
Use the Move Function dialog box to move the currently selected function to a new position in the current worksheet.

When you set the row and column increments, the move process automatically updates cell references by these values.

- Positive increments move rows down and columns to the right.
- Negative increments move rows up and columns to the left.

You can also optionally move the cell contents of any ranges referenced by the function.

Here is an illustration of the Move Function dialog box, set to move the location by two rows and two columns, and to move the cell contents:



# Function Reference

---

componentinfo  
deploytool  
mcc

# componentinfo

---

## Purpose

Query system registry about component created with the MATLAB® Builder™ EX product

## Syntax

```
info = componentinfo
info = componentinfo(component_name)
info = componentinfo(component_name, major_revision_number)
info = componentinfo(component_name, major_revision_number,
    minor_revision_number)
```

## Arguments

<i>component_name</i>	The MATLAB® string providing the name of a MATLAB Builder EX component. Names are case sensitive. If this argument is not supplied, the function returns information on all installed components.
<i>major_revision_number</i>	Component major revision number. If this argument is not supplied, the function returns information on all major revisions.
<i>minor_revision_number</i>	Component minor revision number. Default value is 0.

## Description

`info = componentinfo` returns information for all components installed on the system.

`info = componentinfo(component_name)` returns information for all revisions of *component\_name*.

`info = componentinfo(component_name, major_revision_number)` returns information for the most recent minor revision corresponding to *major\_revision\_number* of *component\_name*.

`info = componentinfo(component_name, major_revision_number, minor_revision_number)` returns information for the specific major and minor version of *component\_name*.

The return value is an array of structures representing all the registry and type information needed to load and use the component.

When you supply a component name, *major\_revision\_number* and *minor\_revision\_number* are interpreted as shown below.

Value	Information Returned
> 0	Information on a specific major and minor revision
0	Information on the most recent revision. When omitted, <i>minor_revision_number</i> is assumed to be equal to 0.
< 0	Information on all versions

---

**Note** Although properties and events may appear in the output for `componentinfo`, they are not supported by builder components.

---

## Registry Information

The information about a component has the fields shown in the following table.

### Registry Information Returned by `componentinfo`

Field	Description
Name	Component name
TypeLib	Component type library
LIBID	Component type library GUID
MajorRev	Major version number
MinorRev	Minor version number
FileName	Type library file name and path. Since all the MATLAB Builder EX components have the type library bound into the DLL, this file name is the same as the DLL name and path.

## Registry Information Returned by componentinfo (Continued)

Field	Description
Interfaces	<p>An array of structures defining all interface definitions in the type library. Each structure contains two fields:</p> <ul style="list-style-type: none"><li>• Name - Interface name</li><li>• IID - Interface GUID</li></ul>
CoClasses	<p>An array of structures defining all COM classes in the component. Each structure contains these fields:</p> <ul style="list-style-type: none"><li>• Name - Class name</li><li>• CLSID - GUID of the class</li><li>• ProgID - Version dependent program ID</li><li>• VerIndProgID - Version independent program ID</li><li>• InprocServer32 - Full name and path to component DLL</li><li>• Methods - A structure containing function prototypes of all class methods defined for this interface. This structure contains four fields:<ul style="list-style-type: none"><li>▪ IDL - An array of Interface Description Language function prototypes</li><li>▪ M - An array of MATLAB function prototypes</li><li>▪ C - An array of C-language function prototypes</li><li>▪ VB - An array of VBA function prototypes</li></ul></li><li>• Properties - A cell array containing the names of all class properties.</li><li>• Events - A structure containing function prototypes of all events defined for this class. This structure contains four fields:</li></ul>



## Examples

Function Call	Returns
<code>Info = componentinfo</code>	Information for all installed components.
<code>Info = componentinfo('mycomponent')</code>	Information for all revisions of mycomponent.
<code>Info = componentinfo('mycomponent',1,0)</code>	Information for revision 1.0 of mycomponent.

# deploytool

---

<b>Purpose</b>	Open GUI for the MATLAB® Builder™ EX and the MATLAB® Compiler™ products
<b>Syntax</b>	deploytool
<b>Description</b>	<p>The <code>deploytool</code> command displays the Deployment Tool dialog box, which is the graphical user interface (GUI) for the MATLAB Builder EX and the MATLAB Compiler products.</p> <p>See “Creating and Building a Component” on page 1-4 for more information about using the Deployment Tool to create COM components, and see the MATLAB Compiler documentation for information about using the Deployment Tool to create standalone applications and libraries.</p>
<b>See Also</b>	<p>“Product Overview” on page 1-2</p> <p>Chapter 2, “Programming with the MATLAB® Builder™ EX Product”</p>

**Purpose** Invoke MATLAB® Compiler™

**Syntax**

```
mcc -W 'excel:component_name,class_name,major.minor'
[-b] [-T link:lib file1..[filen]]
[-d output_dir_path]
```

```
mcc -B 'cexcel:component_name,class_name,major.minor'
[-d output_dir_path]
```

**Description** `mcc` is the MATLAB® command that invokes the MATLAB Compiler product. You can issue the `mcc` command either from the MATLAB command prompt (MATLAB mode) or the DOS or UNIX® command line (standalone mode).

**Options** The `-W` option is used when running `mcc` with the builder.

---

**Note** For a complete list of all `mcc` command options, see `mcc` in the MATLAB Compiler User's Guide documentation.

---

`-W`

Tells the compiler to create an Excel® wrapper. This option takes a string argument that specifies the following characteristics of the component.

<b>-W String Elements</b>	<b>Description</b>
<code>excel:</code>	Keyword that tells the compiler the type of component to create, followed by a colon. Specify <code>excel</code> to create an Excel component.
<code>component_name</code>	Specifies the name of the component to be created.

<b>-W String Elements</b>	<b>Description</b>
<i>class_name</i>	Specifies the name of the class to be created. If you do not specify the class name, <i>mcc</i> uses the component name as the default.
<i>major</i>	Specifies the major version number (for example, 1 in 1.0). If you do not specify a version number, <i>mcc</i> uses the latest version built or 1.0, if there is no previous version.
<i>minor</i>	Specifies the minor version number (for example, 0 in 1.0). If you do not specify a version number, <i>mcc</i> uses the latest version built or 1.0, if there is no previous version.

`[-d output_dir_path]`

(Optional) Tells builder to create a directory and copy the output files to it. If you use *mcc* instead of the Deployment Tool, the *project\_directory\src* and *project\_directory\distrib* directories are not automatically created.

`[-T link:lib file1..filen]`

(Optional) Tells the compiler to create a DLL. Specify the keyword `link:lib`, which links objects into a shared library (DLL).

`[-b]`

(Optional) Generates an Excel-compatible formula function for each M-file.

`-B`

Tells the compiler to use the `cexcel` bundle file option to simplify command line entry. By using this alternative to the `-W` option, the `-T` or the `-b` options do not need to be specified.

<b>-B String Elements</b>	<b>Description</b>
<code>cexcel:</code>	Keyword that tells the compiler to create an Excel component using the bundle file option.

<b>-B String Elements</b>	<b>Description</b>
<i>component_name</i>	Specifies the name of the component to be created.
<i>class_name</i>	Specifies the name of the class to be created. If you do not specify the class name, <code>mcc</code> uses the component name as the default.
<i>major</i>	Specifies the major version number (for example, 1 in 1.0). If you do not specify a version number, <code>mcc</code> uses the latest version built or 1.0, if there is no previous version.
<i>minor</i>	Specifies the minor version number (for example, 0 in 1.0). If you do not specify a version number, <code>mcc</code> uses the latest version built or 1.0, if there is no previous version.

`[-d output_dir_path]`

(Optional) Tells builder to create a directory and copy the output files to it. If you use `mcc` instead of the Deployment Tool, the `project_directory\src` and `project_directory\distrib` directories are not automatically created.

## Examples

### Using -W To Create an Excel Component

```
mcc -W 'excel:mycomponent,myclass,1.0' -T link:lib foo.m bar.m
```

This example shows the `mcc` command used to create a COM component called `mycomponent` containing single COM class named `myclass` with methods `foo` and `bar`, and a version of 1.0 (note both major and minor versions are coded). The `-T` option tells `mcc` to create a DLL.

### Using -b To Create a Function For Each M-file

```
mcc -W 'excel:mycomponent,myclass,1.0' -b -T link:lib foo.m bar.m
```

To generate an Excel-compatible formula function for each M-file, specify the `-b` option.

### **Using -B To Simplify Command Input**

```
mcc -B 'cexcel:mycomponent,myclass,1.0' foo.m bar.m
```

As an alternative to using the `excel` keyword, use the `cexcel` bundle file option to simplify command line input. In the example, note how you do not need to specify the `-T` or the `-b` options when using `-B`.

# Utility Library for Microsoft® COM Components

---

Referencing the Utility Classes  
(p. 6-2)

Utility Library Classes (p. 6-3)

Enumerations (p. 6-31)

Referencing the classes in your  
programming environment

Describes the classes provided in the  
Utility Library.

Describes the three provided sets of  
constants.

## Referencing the 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 the MATLAB® Builder™ EX product.

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

```
mwregsvr mwcomutil.dll
```

The MWComUtil library includes seven classes (see “Utility Library Classes” on page 6-3) and three enumerated types (see “Enumerations” on page 6-31). Before using these types, you must make explicit references to the MWComUtil type libraries in the Microsoft® Visual Basic® IDE. To do this select **Tools>References** from the main menu of the VB editor. The References dialog box appears with a scrollable list of available type libraries. From this list select **MWComUtil 1.0 Type Library** and click **OK**.

---

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

---



## Utility Library Classes

### In this section...

- “Class MWUtil” on page 6-3
- “Class MWFlags” on page 6-10
- “Class MWStruct” on page 6-16
- “Class MWField” on page 6-23
- “Class MWComplex” on page 6-24
- “Class MWSparse” on page 6-26
- “Class MWArg” on page 6-29

### 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

- “Sub MWInitApplication(pApp As Object)” on page 6-3
- “Sub MWPack(pVarArg, [Var0], [Var1], ... , [Var31])” on page 6-5
- “Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoSize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])” on page 6-6
- “Sub MWDate2VariantDate(pVar)” on page 6-8

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 the MATLAB® Builder for .NET product. An error is generated if a method call is made to a member class of any MATLAB Builder for .NET 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

```

**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 nonempty and nonmissing. (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 Variants 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], [bAutoSize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])**

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

**Parameters.**

Argument	Type	Description
VarArg	Variant	Input array of Variants to be processed

Argument	Type	Description
nStartAt	Long	Optional starting index (zero-based) in the array to begin processing. Default = 0.
bAutoSize	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.** The MATLAB product 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 Doubles 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("MComUtil.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 “Data Conversion Rules ” for more information on conversion between the MATLAB product and COM Automation types). All MATLAB Builder for .NET COM components contain a reference to an MWFlags object that can modify data conversion rules at the object level. This class contains these properties:

- “Property ArrayFormatFlags As MWArrayFormatFlags” on page 6-10
- “Property DataConversionFlags As MWDataConversionFlags” on page 6-13
- “Sub Clone(ppFlags As MWFlags)” on page 6-15

### **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 6-11
- “Property InputArrayIndFlag As Long” on page 6-11
- “Property OutputArrayFormat As mwArrayFormat” on page 6-12
- “Property OutputArrayIndFlag As Long” on page 6-12
- “Property AutoResizeOutput As Boolean” on page 6-13
- “Property TransposeOutput As Boolean” on page 6-13



**Property InputArrayFormat As mwArrayFormat.** This property of type `mwArrayFormat` controls the formatting of arrays passed as input parameters to .NET Builder 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
<code>mwArrayFormatAsIs</code>	Converts arrays according to the default conversion rules listed in “Data Conversion Rules ” on page A-2.
<code>mwArrayFormatCell</code>	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.
<code>mwArrayFormatMatrix</code>	Coerces all arrays into matrices. When an input argument is encountered that is an array of Variants (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 Variants 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 .NET Builder 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
<code>mwArrayFormatAsIs</code>	Converts arrays according to the default conversion rules listed in “Data Conversion Rules ” on page A-2.
<code>mwArrayFormatMatrix</code>	Coerces all arrays into matrices. When an output cell array argument is encountered (the default behavior converts it to an array of <code>Variants</code> ), the data converter converts this array to a <code>Variant</code> 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 <code>Variants</code> is created.
<code>mwArrayFormatCell</code>	Coerces all output arrays into arrays of <code>Variants</code> . Output scalar or numeric array arguments are converted to arrays of <code>Variants</code> , each <code>Variant</code> containing a scalar value for the respective index.

**Property OutputArrayIndFlag As Long.** This property is similar to the `InputArrayIndFlag` 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 `AutoSizeOutput 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 6-13
- “Property `InputDateFormat As mwDateFormat`” on page 6-13
- “Example” on page 6-14
- “Property `OutputAsDate As Boolean`” on page 6-15
- “Property `DateBias As Long`” on page 6-15

**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 “Data Conversion Rules ” on page A-2.

**Property `InputDateFormat As mwDateFormat`.** This property converts dates passed as input parameters to method calls on .NET Builder 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 “Data Conversion Rules ” on page A-2.
mwDateFormatString	Convert input dates to strings.

**Example.** 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 `AutoSizeOutput` 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

```

**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 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.

**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 created by MATLAB® Builder™ NE. To process dates with such code, set this property to 0.

### 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:

- “Sub Initialize([varDims], [varFieldNames])” on page 6-16
- “Property Item([i0], [i1], ..., [i31]) As MWField” on page 6-18
- “Property NumberOfFields As Long” on page 6-20
- “Property NumberOfDims As Long” on page 6-20
- “Property Dims As Variant” on page 6-21
- “Property FieldNames As Variant” on page 6-21
- “Sub Clone(ppStruct As MWStruct)” on page 6-22

### **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.**

<b>Argument</b>	<b>Type</b>	<b>Description</b>
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 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
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:

- “Property Name As String” on page 6-23
- “Property Value As Variant” on page 6-23
- “Property MWFlags As MWFlags” on page 6-23
- “Sub Clone(ppField As MWField)” on page 6-23

### **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:

- “Property Real As Variant” on page 6-24
- “Property Imag As Variant” on page 6-24
- “Property MWFlags As MWFlags” on page 6-25
- “Sub Clone(ppComplex As MWComplex)” on page 6-26

**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 nonempty 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:

- “Property NumRows As Long” on page 6-26
- “Property NumColumns As Long” on page 6-27
- “PropertyRowIndex As Variant” on page 6-27
- “Property ColumnIndex As Variant” on page 6-27
- “Property Array As Variant” on page 6-27
- “Property MWFlags As MWFlags” on page 6-27
- “Sub Clone(ppSparse As MWSparse)” on page 6-28

**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 theRowIndex 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:

- “Property Value As Variant” on page 6-30
- “Property MWFlags As MWFlags” on page 6-30
- “Sub Clone(ppArg As MWArg)” on page 6-30

### 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.

## Enumerations

In this section...
“Enum mwArrayFormat” on page 6-31
“Enum mwDataType” on page 6-31
“Enum mwDateFormat” on page 6-32

### 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
<code>mwArrayFormatAsIs</code>	0	Do not reformat the array.
<code>mwArrayFormatMatrix</code>	1	Format the array as a matrix.
<code>mwArrayFormatCell</code>	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
<code>mwTypeDefault</code>	0	N/A
<code>mwTypeLogical</code>	3	logical
<code>mwTypeChar</code>	4	char
<code>mwTypeDouble</code>	6	double

**mwDataType Values (Continued)**

<b>Constant</b>	<b>Numeric Value</b>	<b>MATLAB Type</b>
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**

<b>Constant</b>	<b>Numeric Value</b>	<b>Description</b>
mwDateFormatNumeric	0	Format dates as numeric values
mwDateFormatString	1	Format dates as strings

# Data Conversion

---

Data Conversion Rules (p. A-2)	Describes the process of converting data between the MATLAB® product and Microsoft® COM variants
Array Formatting Flags (p. A-12)	Describes the flags that control the formatting of data
Data Conversion Flags (p. A-14)	Describes the flags that control the conversion of data

## Data Conversion Rules

This topic describes the data conversion rules for the MATLAB® Builder™ EX components. These components are dual interface Microsoft® 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.

---

When a method is invoked on an MATLAB Builder EX component, the input parameters are converted to the MATLAB® internal array format and passed to the compiled MATLAB function. When the function exits, the output parameters are converted from the 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 Application Program Interface (API) provides many functions for creating and manipulating VARIANTS in C/C++, and Visual Basic® provides native language support for this type.

---

**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 the 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.



The following table lists the VARIANT type codes supported by the MATLAB Builder EX components.

### VARIANT Type Codes Supported

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual BasicType	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 <sup>+</sup>	vbCurrency	Currency	Currency value (64-bit integer, scaled by 10,000)
VT_BSTR	BSTR <sup>+</sup>	vbString	String	String value
VT_ERROR	SCODE <sup>+</sup>	vbError	—	A HRESULT (Signed four-byte integer representing a COM error code)

**VARIANT Type Codes Supported (Continued)**

<b>VARIANT Type Code (C/C++)</b>	<b>C/C++ Type</b>	<b>Variant Type Code (Visual Basic)</b>	<b>Visual BasicType</b>	<b>Definition</b>
VT_DATE	DATE <sup>+</sup>	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 <sup>+</sup>	vbDecimal	—	96-bit (12-byte) unsigned integer, scaled by a variable power of 10
VT_BOOL	VARIANT_BOOL <sup>+</sup>	vbBoolean	Boolean	Two-byte Boolean value (0xFFFF = True; 0x0000 = False)
VT_DISPATCH	IDispatch*	vbObject	Object	IDispatch* pointer to an object
VT_VARIANT	VARIANT <sup>+</sup>	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

**VARIANT Type Codes Supported (Continued)**

<b>VARIANT Type Code (C/C++)</b>	<b>C/C++ Type</b>	<b>Variant Type Code (Visual Basic)</b>	<b>Visual BasicType</b>	<b>Definition</b>
<anything> VT_BYREF	—	—	—	Bitwise combine VT_BYREF with any basic type to declare as a reference to a value

+ Denotes Windows-specific type. Not part of standard C/C++.

The following table lists the rules for converting from the MATLAB product to COM.

**MATLAB® to COM VARIANT Conversion Rules**

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
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 B-16.) This object is passed as a VT_DISPATCH type.

**MATLAB® to COM VARIANT Conversion Rules (Continued)**

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
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 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.	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.
sparse	VT_DISPAATCH	VT_DISPATCH	A MATLAB sparse array is converted to an MWSparse object. (See “Class MWSparse” on page B-27.) This object is passed as a VT_DISPATCH type.

**MATLAB® to COM VARIANT Conversion Rules (Continued)**

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
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 M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)
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 M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)
int8	A real 1-by-1 int8 matrix converts to a VARIANT of type VT_I1. A complex 1-by-1 int8 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional int8 matrix converts to a VARIANT of type VT_I1 VT_ARRAY. A complex multidimensional int8 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)

**MATLAB® to COM VARIANT Conversion Rules (Continued)**

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
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 M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)
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 M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)
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 M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)

**MATLAB® to COM VARIANT Conversion Rules (Continued)**

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
int32	A 1-by-1 int32 matrix converts to a VARIANT of type VT_I4. A complex 1-by-1 int32 matrix converts to a VARIANT of type VT_DISPATCH.	A multidimensional int32 matrix converts to a VARIANT of type VT_I4 VT_ARRAY. A complex multidimensional int32 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)
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 M-functions using the MWComplex class. (See “Class MWComplex” on page B-25.)
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_Boolean	VT_Boolean VT_ARRAY	

The following table lists the rules for conversion from COM to MATLAB.

**COM VARIANT to MATLAB® Conversion Rules**

<b>VARIANT Type</b>	<b>MATLAB Data Type (scalar or array data)</b>	<b>Comments</b>
VT_EMPTY	N/A	Empty array created.

**COM VARIANT to MATLAB® Conversion Rules (Continued)**

<b>VARIANT Type</b>	<b>MATLAB Data Type (scalar or array data)</b>	<b>Comments</b>
VT_I1	int8	
VT_UI1	uint8	
VT_I2	int16	
VT_UI2	uint16	
VT_I4	int32	
VT_UI4	uint32	
VT_R4	single	
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	<ol style="list-style-type: none"> <li>VARIANT dates are stored as doubles starting at midnight Dec. 31, 1899. The 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.</li> <li>VARIANT dates can be optionally converted to strings. See “Data Conversion Flags” on page A-14 for more information on type coercion.</li> </ol>
VT_INT	int32	



**COM VARIANT to MATLAB® Conversion Rules (Continued)**

<b>VARIANT Type</b>	<b>MATLAB Data Type (scalar or array data)</b>	<b>Comments</b>
VT_UINT	uint32	
VT_DECIMAL	double	
VT_BOOL	logical	
VT_DISPATCH	(varies)	<p>IDispatch* pointers are treated within the context of what they point to. Objects must be supported types with known 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 upon the rules for the particular VARIANT obtained.</p> <p>Currently, support exists for Excel® Range objects as well as the MATLAB Builder EX types MWStruct, MWComplex, MWSparse, and MWArg. See “Utility Library Classes” on page B-3 for information on the MATLAB Builder EX types.</p>
<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 MATLAB® Builder™ EX 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 on page A-5 and COM VARIANT to MATLAB® Conversion Rules on page A-9. 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	<p>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.</p> <p>Valid values for this flag are <code>mwArrayFormatAsIs</code>, <code>mwArrayFormatMatrix</code>, and <code>mwArrayFormatCell</code>.</p> <p><code>mwArrayFormatAsIs</code> passes the array unchanged.</p> <p><code>mwArrayFormatMatrix</code> (default) formats all arrays as matrices. When the input VARIANT is of type <code>VT_ARRAY   type</code>, where <code>type</code> is any numeric type, this flag has no effect. When the input VARIANT is of type <code>VT_VARIANT   VT_ARRAY</code>, VARIANTS 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.</p> <p><code>mwArrayFormatCell</code> interprets all arrays as MATLAB cell arrays.</p>

**Array Formatting Flags (Continued)**

<b>Flag</b>	<b>Description</b>
InputArrayIndFlag	Sets the input array indirection level used with the InputArrayFormat flag (applicable only to nested arrays, i.e., VARIANT arrays of VARIANTS, which themselves are arrays). The default value for this flag is zero, which applies the InputArrayFormat flag to the outermost 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.
AutoSizeOutput	(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 an MATLAB Builder EX component from Excel where the MATLAB function returns outputs as row vectors, and you want the data in columns.

## Data Conversion Flags

In this section...
“CoerceNumericToType” on page A-14
“InputDateFormat” on page A-15
“OutputAsDate As Boolean” on page A-16
“DateBias As Long” on page A-16

### **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

VT\_DATE

Valid values for this flag are

`mwTypeDefault`

`mwTypeChar`

`mwTypeDouble`

`mwTypeSingle`

`mwTypeLogical`

`mwTypeInt8`

`mwTypeUInt8`

`mwTypeInt16`

`mwTypeUInt16`

`mwTypeInt32`

`mwTypeUInt32`

The default for this flag, `mwTypeDefault`, converts numeric data according to the rules listed in “Data Conversion Rules ” on page A-2.

## **InputDateFormat**

This flag tells the data converter how to convert VARIANT dates to the 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 on page A-3. 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 the MATLAB numeric date conversions. The default value of this property is 693960, which represents the difference between the COM Date type and the 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 MATLAB® Builder™ EX components. To process dates with such code, set this property to 0.

# Utility Library

---

Referencing Utility Classes (p. B-2)	How to reference the classes in your programming environment.
Utility Library Classes (p. B-3)	Describes the classes provided in the Utility library.
Enumerations (p. B-32)	Describes the sets of constants provided with the library.

## Referencing Utility Classes

This section describes the `MWComUtil` library provided with MATLAB® Builder™ EX. 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 builder components.

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

```
mwregsvr mwcomutil.dll
```

The `MWComUtil` library includes seven classes (see “Utility Library Classes” on page B-3) and three enumerated types (see “Enumerations” on page B-32). Before using these types, you must make explicit references to the `MWComUtil` type libraries in the Microsoft® Visual Basic® IDE. To do this, click **Tools > References** from the main menu of the VB editor. The References dialog box appears with a scrollable list of available type libraries. From this list, select **MWComUtil x.x Type Library** (where *x.x* is the version number of the MCR) and click **OK**.

---

**Note** To obtain the MCR version number, use the MATLAB function `mcrversion`.

---

---

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

---



## Utility Library Classes

### In this section...

“Class MWUtil” on page B-3

“Class MWFlags” on page B-10

“Class MWStruct” on page B-16

“Class MWField” on page B-24

“Class MWComplex” on page B-25

“Class MWSparse” on page B-27

“Class MWArg” on page B-30

### 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 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

- “Sub `MWInitApplication(pApp As Object)`” on page B-3
- “Sub `MWPack(pVarArg, [Var0], [Var1], ... , [Var31])`” on page B-5
- “Sub `MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])`” on page B-6
- “Sub `MWDate2VariantDate(pVar)`” on page B-8

The function prototypes use Visual Basic® syntax.

### Sub `MWInitApplication(pApp As Object)`

Initializes the library with the current instance of 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 builder components. An error is generated if a method call is made to a member class of any builder 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
```

**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 nonempty and nonmissing. (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 Variants to pack into the array. 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], [bAutoSize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])**

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

#### **Parameters.**

<b>Argument</b>	<b>Type</b>	<b>Description</b>
VarArg	Variant	Input array of Variants to be processed

Argument	Type	Description
nStartAt	Long	Optional starting index (zero-based) in the array to begin processing. Default = 0.
bAutoSize	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. 0 to 32 arguments can be passed.

**Return Value.** None.

**Remarks.** This function can process a Variant array in a 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 *i*th 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 (see “Data Conversion Rules” on page A-2 for more information on conversion between MATLAB and COM date values). 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 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("MComUtil.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 “Data Conversion Rules” on page A-2 for more information on conversion between MATLAB and COM Automation types). All builder components contain a reference to an MWFlags object that can modify data conversion rules at the object level. This class contains these properties:

- “Property ArrayFormatFlags As MWArrayFormatFlags” on page B-10
- “Property DataConversionFlags As MWDataConversionFlags” on page B-13
- “Sub Clone(ppFlags As MWFlags)” on page B-15

## **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 these properties:

- “Property InputArrayFormat As mwArrayFormat” on page B-11
- “Property InputArrayIndFlag As Long” on page B-11
- “Property OutputArrayFormat As mwArrayFormat” on page B-12
- “Property OutputArrayIndFlag As Long” on page B-12
- “Property AutoResizeOutput As Boolean” on page B-13
- “Property TransposeOutput As Boolean” on page B-13



**Property InputArrayFormat As mwArrayFormat.** This property of type `mwArrayFormat` controls the formatting of arrays passed as input parameters to builder class methods. The default value is `mwArrayFormatMatrix`. The behaviors indicated by this flag are listed in the following table.

### Array Formatting Rules for Input Arrays

Value	Behavior
<code>mwArrayFormatAsIs</code>	Converts arrays according to the default conversion rules listed in “Data Conversion Rules ” on page A-2.
<code>mwArrayFormatCell</code>	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.
<code>mwArrayFormatMatrix</code>	Coerces all arrays into matrices. When an input argument is encountered that is an array of Variants (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 Variants 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 mxArrayFormat.** This property of type `mxArrayFormat` controls the formatting of arrays passed as output parameters to builder class methods. The default value is `mxArrayFormatAsIs`. The behaviors indicated by this flag are listed in the following table.

**Array Formatting Rules for Output Arrays**

Value	Behavior
<code>mxArrayFormatAsIs</code>	Converts arrays according to the default conversion rules listed in MATLAB® to COM VARIANT Conversion Rules on page A-5.
<code>mxArrayFormatMatrix</code>	Coerces all arrays into matrices. When an output cell array argument is encountered (the default behavior converts it to an array of Variants), 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 Variants is created.
<code>mxArrayFormatCell</code>	Coerces all output arrays into arrays of Variants. Output scalar or numeric array arguments are converted to arrays of Variants, each Variant containing a scalar value for the respective index.

**Property OutputArrayIndFlag As Long.** This property is similar to the `InputArrayIndFlag` 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 `AutoSizeOutput 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 an builder 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 B-13
- “Property `InputDateFormat As mwDateFormat`” on page B-13
- “Property `OutputAsDate As Boolean`” on page B-15
- “Property `DateBias As Long`” on page B-15

**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 `COM VARIANT` to MATLAB® Conversion Rules on page A-9.

**Property `InputDateFormat As mwDateFormat`.** This property converts dates passed as input parameters to method calls on builder 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 COM VARIANT to MATLAB® Conversion Rules on page A-9.
mwDateFormatString	Convert input dates to strings.

**Example.** 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>0);
```

This function produces a row vector of all the prime numbers from 0 to 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
```

**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 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.

**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 builder components. To process dates with such code, set this property to 0.

### **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 these properties/methods:

- “Sub Initialize([varDims], [varFieldNames])” on page B-16
- “Property Item([i0], [i1], ..., [i31]) As MWField” on page B-18
- “Property NumberOfFields As Long” on page B-21
- “Property NumberOfDims As Long” on page B-21
- “Property Dims As Variant” on page B-21
- “Property FieldNames As Variant” on page B-21
- “Sub Clone(ppStruct As MWStruct)” on page B-22

**Sub Initialize([varDims], [varFieldNames])**

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 and get the value of a field at a particular index in the structure array.

#### **Parameters.**

<b>Argument</b>	<b>Type</b>	<b>Description</b>
i0,i1, ..., i31	Variant	Optional index arguments. 0 to 32 index arguments can be entered. To reference an element of 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 a 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 following 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 structure 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 structure 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 these properties/methods:

- “Property Name As String” on page B-24
- “Property Value As Variant” on page B-24
- “Property MWFlags As MWFlags” on page B-24
- “Sub Clone(ppField As MWField)” on page B-24

### **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 these properties/methods:

- “Property Real As Variant” on page B-25
- “Property Imag As Variant” on page B-25
- “Property MWFlags As MWFlags” on page B-26
- “Sub Clone(ppComplex As MWComplex)” on page B-27

**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 nonempty 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 these properties/methods:

- “Property NumRows As Long” on page B-27
- “Property NumColumns As Long” on page B-28
- “PropertyRowIndex As Variant” on page B-28
- “Property ColumnIndex As Variant” on page B-28
- “Property Array As Variant” on page B-28
- “Property MWFlags As MWFlags” on page B-28
- “Sub Clone(ppSparse As MWSparse)” on page B-29

### 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 theRowIndex array.

### **Property NumColumns As Long**

Stores the column dimension for the array. The value of NumColumns must be nonnegative. If the value is 0, 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 MWSpase 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 MWSparsed
    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 these properties/methods:

- “Property Value As Variant” on page B-31
- “Property MWFlags As MWFlags” on page B-31
- “Sub Clone(ppArg As MWArg)” on page B-31

### 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.

## Enumerations

In this section...
“Enum mwArrayFormat” on page B-32
“Enum mwDataType” on page B-32
“Enum mwDateFormat” on page B-33

### Enum mwArrayFormat

The `mwArrayFormat` enumeration is a set of constants that denote an array formatting rule for data conversion. The following table lists the members of this enumeration.

#### mwArrayFormat Values

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

### Enum mwDataType

The `mwDataType` enumeration is a set of constants that denote a MATLAB® numeric type. The following table lists the members of this enumeration.

#### mwDataType Values

Constant	Numeric Value	MATLAB Type
<code>mwTypeDefault</code>	0	N/A
<code>mwTypeLogical</code>	3	logical
<code>mwTypeChar</code>	4	char

**mwDataType Values (Continued)**

<b>Constant</b>	<b>Numeric Value</b>	<b>MATLAB Type</b>
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. The following table lists the members of this enumeration.

**mwDateFormat Values**

<b>Constant</b>	<b>Numeric Value</b>	<b>Description</b>
mwDateFormatNumeric	0	Format dates as numeric values.
mwDateFormatString	1	Format dates as strings.





# Troubleshooting

---

This appendix provides a table showing errors you may encounter using the MATLAB® Builder™ EX product, probable causes for these errors, and suggested solutions.

---

**Note** The MATLAB Builder EX product uses the MATLAB® Compiler™ product to generate components. This means that you might see diagnostic messages from the MATLAB Compiler product. See the MATLAB Compiler documentation for more information about those messages.

---

### MATLAB® Builder™ EX Errors and Suggested Solutions

Message	Probable Cause	Suggested Solution
MBUILD.BAT: Error: The chosen compiler does not support building COM objects.	The chosen compiler does not support building COM objects.	Rerun <code>mbuild -setup</code> and choose a supported compiler.
Error in <code>component_name.class_name</code> : Error getting data conversion flags.	Usually caused by <code>mwcomutil.dll</code> not being registered.	Open a DOS window, change directories to <code>matlabroot\bin\win32</code> ( <code>matlabroot</code> represents the location of the MATLAB® product on your system), and run the command <code>mwregsvr mwcomutil.dll</code> .
Error in VBAProject: ActiveX component can't create object.	<ul style="list-style-type: none"> <li>Project DLL is not registered.</li> <li>An incompatible MATLAB DLL exists somewhere on the system path.</li> </ul>	If the DLL is not registered, open a DOS window, change directories to <code>&lt;projectdir&gt;\distrib</code> ( <code>&lt;projectdir&gt;</code> represents the location of your project files), and run the command: <code>mwregsvr &lt;projectdll&gt;.dll</code> .

**MATLAB® Builder™ EX Errors and Suggested Solutions (Continued)**

<b>Message</b>	<b>Probable Cause</b>	<b>Suggested Solution</b>
Error in VBAProject: Automation error The specified module could not be found.	This usually occurs if the MATLAB product is not on the system path. This error message occurs if you have more than one version of the MATLAB product on your system path.	Anytime you have multiple versions of the MATLAB product, ensure that the newest version of MATLAB appears on your path first. You can verify that the newest version of the MATLAB product is on the path first by typing path at the DOS prompt. See Required Locations to Develop and Use Components on page C-4.
LoadLibrary ("component_name.dll") failed - The specified module could not be found.	You may get this error message while registering the project DLL from the DOS prompt. This usually occurs if the MATLAB product is not on the system path.	See Required Locations to Develop and Use Components on page C-4.
Cannot recompile the M file xxxx because it is already in the library libmmfile.mlib.	The name you have chosen for your M-file duplicates the name of an M-file already in the library of precompiled M-files.	Rename the M-file, choosing a name that does not duplicate the name of an M-file already in the library of precompiled M-files.
Arguments may only be defaulted at the end of an argument list.	You have modified the VB script generated for the MATLAB Builder EX product and have not provided one or more arguments used in the modified script.	Provide a value for any argument that requires an explicit value. Arguments that accept defaults appear at the end of the argument list.

### Required Locations to Develop and Use Components

Component	Development Machine	Target Machine
MCR	Make sure that <i>matlabroot\bin\win32</i> appears on your system path ahead of any other the MATLAB product installations. ( <i>matlabroot</i> is your root MATLAB directory.)	Verify that <i>mcr_root\ver\runtime\win32</i> appears on your system path. ( <i>mcr_root</i> is your root MCR directory.)
CTF	Verify that the CTF file is in the same directory as your program's executable file.	

## Microsoft® Excel® Errors and Suggested Solutions

Message	Probable Cause	Suggested Solution
<p>The macros in this project are disabled. Please refer to the online help or documentation of the host application to determine how to enable macros.</p> <p>Note: Wording may vary depending upon the version of Excel® you are running.</p>	<p>The macro security for Excel is set to High.</p>	<p>Set Excel macro security to Medium on the <b>Security Level</b> tab. Select <b>Tools &gt; Macro &gt; Security</b>.</p>

## Function Wizard Problems

Problem	Probable Cause	Suggested Solution
<p>The Function Wizard Help does not appear.</p>	<p>The Function Wizard Help file (mlfunction.chm) is not in the same directory as the Function Wizard add-in (mlfunction.xla).</p>	<p>Copy the Help file (mlfunction.chm) into the same directory as the add-in.</p>



# Examples

---

Use this list to find examples in the documentation.

## **Calling a MATLAB Function from Microsoft® Excel®**

“Magic Square Examples” on page 3-2

## **Using Multiple Files and Variable Arguments**

“Multiple Files and Variable Arguments Example” on page 3-6

## **Creating a Comprehensive Microsoft® Excel® Add-In: Spectral Analysis**

“Spectral Analysis Example” on page 3-13



## A

- Add-ins
  - permission to build and deploy 2-15
- array formatting flags 2-18

## C

- class 1-2
- class method
  - calling 2-7
- Class MWFlags 6-10 B-10
- Class MWUtil 6-3 B-3
- class name 1-2
- COM
  - defined 1-2
- COM component
  - utility classes 6-1
- COM VARIANT A-2
- command line interface 1-6
- compiling
  - complete syntactic details 5-7
- Component Object Model 1-2
- componentinfo function 5-2
- CreateObject function 2-7

## D

- data conversion
  - utility classes for COM components 6-1
- data conversion flags 2-18
- data conversion rules A-2
- deploytool function 5-6
- DLLs
  - utility classes for COM components 6-1

## E

- Enumeration
  - mwArrayFormat 6-31 B-32
  - mwDataType 6-31 B-32

- mwDateFormat 6-32 B-33
- enumerations 6-31 B-32
- errors
  - Excel C-5
  - MATLAB Builder EX C-2

## F

- flags
  - array formatting 2-18
  - data conversion 2-18
- function wizard
  - argument properties 4-14
  - component browser 4-6
  - function properties 4-9
  - function utilities 4-16
  - function viewer 4-6
  - purpose 4-2
- functions 2-4

## M

- mcc
  - syntax 5-7
- methods 1-2
- missing parameter 6-5 B-5
- MWFlags class 6-10 B-10
- MWUtil class 6-3 B-3

## N

- New operator 2-8

## P

- project
  - creating 1-4

## R

- required arguments 4-10

**S**

subroutines 2-4

**T**

troubleshooting

MATLAB Builder EX C-2

**U**

utility library 6-3 B-3

**V**

varargin/varargout 4-10

VARIANT variable A-2