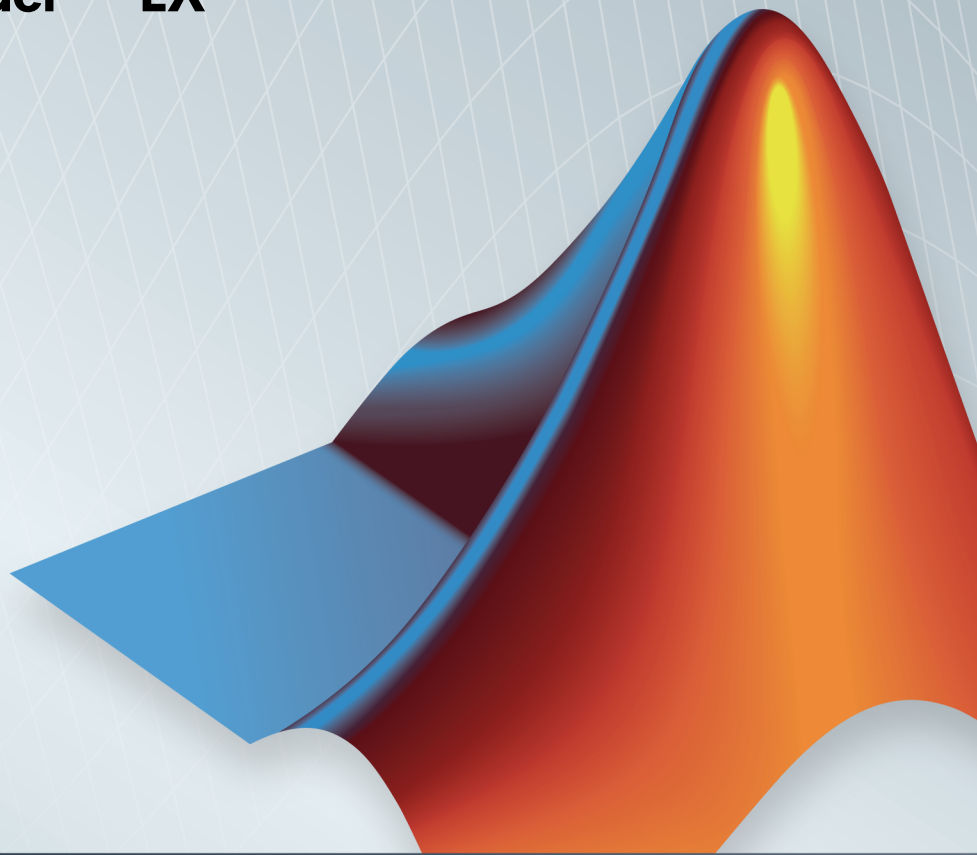


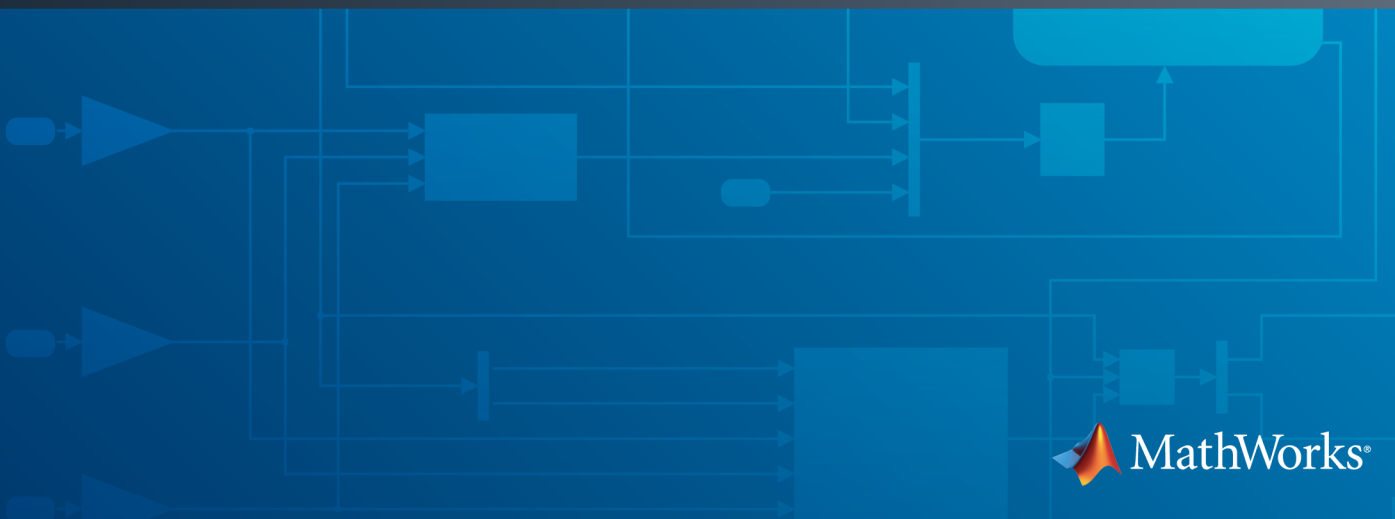
# MATLAB<sup>®</sup> Builder<sup>™</sup> EX

## User's Guide

R2014b



# MATLAB<sup>®</sup>



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*MATLAB<sup>®</sup> Builder<sup>™</sup> EX User's Guide*

© COPYRIGHT 1984–2014 by The MathWorks, Inc.

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

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

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](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**

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 change
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)
October 2008	Online only	Revised for Version 1.2.11 (Release 2008b)
March 2009	Online only	Revised for Version 1.2.12 (Release 2009a)
September 2009	Online only	Revised for Version 1.2.13 (Release 2009b)
March 2010	Online only	Revised for Version 1.2.15 (Release 2010a)
September 2010	Online only	Revised for Version 1.3 (Release 2010b)
April 2011	Online only	Revised for Version 2.0 (Release 2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.3.1 (Release 2013a)
September 2013	Online only	Revised for Version 2.4 (Release 2013b)
March 2014	Online only	Revised for Version 2.5 (Release 2014a)
October 2014	Online only	Revised for Version 2.5.1 (Release 2014b)



## Getting Started

<b>MATLAB Builder EX Product Description</b> .....	<b>1-2</b>
Key Features .....	1-2
<b>MATLAB Application Deployment Products</b> .....	<b>1-3</b>
<b>How MATLAB Builder EX Works</b> .....	<b>1-5</b>
<b>MATLAB Builder EX Prerequisites</b> .....	<b>1-6</b>
Your Role in the Deployment Process .....	1-6
Products, Compilers, and IDE Installation .....	1-7
Macro Execution Security Levels in Microsoft Excel .....	1-7
Deployment Target Architectures and Compatibility .....	1-8
Dependency and Non-Compilable Code Considerations .....	1-9
For More Information .....	1-9
<b>Choosing Function Deployment Workflow</b> .....	<b>1-10</b>
Is Your Function Ready for Deployment? .....	1-10
Other Examples .....	1-10
<b>Create a Microsoft Excel Add-In and COM Component from MATLAB Code</b> .....	<b>1-12</b>
Create an Excel Add-in in MATLAB .....	1-12
<b>Integrate an Add-In and COM Component with Microsoft Excel</b> .....	<b>1-16</b>
Files Necessary for Deployment .....	1-16
Add-in and COM Component Registration .....	1-17
COM Component Incorporation into Microsoft Excel using the Function Wizard .....	1-18
Distributing MATLAB Code Using the MATLAB Runtime .	1-18
Add-in Installation and Distribution .....	1-20

Next Steps .....	1-21
------------------	------

## Customizing a Compiler Project

# 2

<b>Customize the Installer</b> .....	2-2
Change the Application Icon .....	2-2
Add Application Information .....	2-3
Change the Splash Screen .....	2-3
Change the Installation Path .....	2-4
Change the Logo .....	2-4
Edit the Installation Notes .....	2-5
 <b>Manage Required Files in a Compiler Project</b> .....	2-6
Dependency Analysis .....	2-6
Using the Compiler Apps .....	2-6
Using mcc .....	2-6
 <b>Specify Files to Install with the Application</b> .....	2-8
 <b>Manage Support Packages</b> .....	2-9
Using a Compiler App .....	2-9
Using the Command Line .....	2-10

## Integrating with MATLAB Production Server

# 3

<b>Build Deployable Archive and Add-In</b> .....	3-2
 <b>Install the Deployable Archive</b> .....	3-5
 <b>Install the Excel Add-In</b> .....	3-6
Install with the Generated Installer .....	3-6
Install with the Raw Files .....	3-7
 <b>Data Marshaling Rules</b> .....	3-8
Default Marshaling Rules .....	3-8

Change Rules for Marshaling Data into MATLAB . . . . .	3-8
Change Rules for Marshaling Data into Excel . . . . .	3-9
<b>Change the Server Configuration . . . . .</b>	<b>3-10</b>
<b>Enable Client Logging . . . . .</b>	<b>3-12</b>
<b>XLA File Not Generated . . . . .</b>	<b>3-14</b>
<b>Server Configuration Add-in Not Enabled . . . . .</b>	<b>3-15</b>
<b>Error in myfunc: Object reference not set to an object . . . . .</b>	<b>3-16</b>

## The Function Wizard

# 4

<b>Executing Functions and Creating Macros Using the Function Wizard . . . . .</b>	<b>4-2</b>
What Can the Function Wizard Do for Me? . . . . .	4-3
Installation of the Function Wizard . . . . .	4-4
Function Wizard Start-Up . . . . .	4-5
Workflow Selection for MATLAB Functions Ready for Deployment . . . . .	4-6
Defining Functions Ready to Execute . . . . .	4-7
Function Execution . . . . .	4-17
Macro Creation . . . . .	4-18
Macro Execution . . . . .	4-18
Microsoft Visual Basic Code Access (Optional Advanced Task) . . . . .	4-19
For More Information . . . . .	4-21
<b>End-to-End Deployment of a MATLAB Function Using the Function Wizard . . . . .</b>	<b>4-22</b>
What Can the Function Wizard Do for Me? . . . . .	4-3
Example File Copying . . . . .	4-25
mymagic Testing . . . . .	4-25
Installation of the Function Wizard . . . . .	4-4
Function Wizard Start-Up . . . . .	4-27

Workflow Selection for Prototyping and Debugging MATLAB Functions . . . . .	4-28
New MATLAB Function Definition . . . . .	4-30
MATLAB Function Prototyping and Debugging . . . . .	4-41
Function Execution from MATLAB . . . . .	4-43
Microsoft Excel Add-in and Macro Creation Using the Function Wizard . . . . .	4-44
Function Execution from the Deployed Component . . . . .	4-46
Macro Execution . . . . .	4-18
Microsoft Excel Add-in and Macro Packaging using the Function Wizard . . . . .	4-47
Microsoft Visual Basic Code Access (Optional Advanced Task) . . . . .	4-19
For More Information . . . . .	4-49

## MATLAB Code Deployment

# 5

<b>Application Deployment Products and the Compiler Apps . . . . .</b>	<b>5-2</b>
What Is the Difference Between the Compiler Apps and the mcc Command Line? . . . . .	5-2
How Does MATLAB Compiler Software Build My Application? . . . . .	5-2
Dependency Analysis Function . . . . .	5-5
MEX-Files, DLLs, or Shared Libraries . . . . .	5-6
Deployable Archive . . . . .	5-6
<b>Write Deployable MATLAB Code . . . . .</b>	<b>5-10</b>
Compiled Applications Do Not Process MATLAB Files at Runtime . . . . .	5-10
Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files . . . . .	5-11
Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths . . . . .	5-11
Gradually Refactor Applications That Depend on Noncompilable Functions . . . . .	5-12
Do Not Create or Use Nonconstant Static State Variables . . . . .	5-12
Get Proper Licenses for Toolbox Functionality You Want to Deploy . . . . .	5-13



<b>How the Deployment Products Process MATLAB Function Signatures</b> .....	<b>5-14</b>
MATLAB Function Signature .....	5-14
MATLAB Programming Basics .....	5-14
<b>Load MATLAB Libraries using loadlibrary</b> .....	<b>5-16</b>
Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler .....	5-17
<b>Use MATLAB Data Files (MAT Files) in Compiled Applications</b> .....	<b>5-18</b>
Explicitly Including MAT files Using the %#function Pragma .....	5-18
Load and Save Functions .....	5-18
MATLAB Objects .....	5-21

## Microsoft Excel Add-in Creation, Function Execution, and Deployment

# 6

<b>Supported Compilation Targets</b> .....	<b>6-2</b>
Microsoft Excel Add-in .....	6-2
What Are MATLAB Builder EX Components and When Should You Create Them? .....	6-2
MATLAB Builder EX Limitations .....	6-3
<b>The Library Compiler and the Command Line Interface</b> ...	<b>6-4</b>
Using the Library Compiler .....	6-4
Using the mcc Command Line Interface .....	6-4
<b>Add-in and Macro Creation from MATLAB Functions</b> .....	<b>6-5</b>
Create Add-ins and Macros with Single and Multiple Outputs .....	6-5
Work with Variable-Length Inputs and Outputs .....	6-5
<b>Execute Add-in With the Function Wizard</b> .....	<b>6-11</b>
Execute an Add-in to Validate Nongraphical Function Output .....	6-11
Execute a Graphical Function .....	6-11
Create Dialog Box and Error Message Macros .....	6-15

<b>Overview of the Integration Process</b> .....	7-2
<b>Code Your Microsoft Visual Basic Application</b> .....	7-3
When to Use a Formula Function or a Subroutine .....	7-3
Initializing MATLAB Builder EX Libraries with Microsoft Excel .....	7-3
Creating an Instance of a Class .....	7-4
Calling the Methods of a Class Instance .....	7-7
Programming with Variable Arguments .....	7-8
Modifying Flags .....	7-10
Handling Errors During a Method Call .....	7-15
<b>Build and Integrate with Spectral Analysis</b> .....	7-16
Overview .....	7-16
Building the Component .....	7-16
Integrating the Component Using VBA .....	7-18
Testing the Add-in .....	7-25
Packaging and Distributing the Add-in .....	7-27
Installing the Add-in .....	7-29
<b>For More Information</b> .....	7-30

Distribution to End Users

<b>Distribute Your Add-Ins and COM Components to End Users</b> .....	8-2
Distributing MATLAB Code Using the MATLAB Runtime .	8-18
<b>Deploying Your Microsoft Visual Basic Application</b> .....	8-4
Calling Compiled MATLAB Functions from Microsoft Excel .	8-4
Improving Data Access Using the MCR User Data Interface, COM Components, and MATLAB Builder EX .....	8-6
MCR Component Cache and CTF Archive Embedding .....	8-11
MCR Run-Time Options .....	8-13

**Function Reference**

9

**Utility Library for Microsoft COM Components**

10

**Referencing Utility Classes . . . . . 10-2**

**Class MWUtil . . . . . 10-3**

    Sub MWInitApplication(pApp As Object) . . . . . 10-3

    Sub MWInitApplicationWithMCROptions(pApp As Object,  
        [mcrOptionList]) . . . . . 10-5

    Function IsMCRJVMEEnabled() As Boolean . . . . . 10-6

    Function IsMCRInitialized() As Boolean . . . . . 10-6

    Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31]) . . . . . 10-7

    Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As  
        Boolean = False], [pVar0], [pVar1], ..., [pVar31]) . . . . . 10-8

    Sub MWDate2VariantDate(pVar) . . . . . 10-10

**Class MWFlags . . . . . 10-12**

    Property ArrayFormatFlags As MWArrayFormatFlags . . . 10-12

    Property DataConversionFlags As  
        MWDataConversionFlags . . . . . 10-15

    Sub Clone(ppFlags As MWFlags) . . . . . 10-17

**Class MWStruct . . . . . 10-18**

    Sub Initialize([varDims], [varFieldNames]) . . . . . 10-18

    Property Item([i0], [i1], ..., [i31]) As MWField . . . . . 10-19

    Property NumberOfFields As Long . . . . . 10-22

    Property NumberOfDims As Long . . . . . 10-22

    Property Dims As Variant . . . . . 10-22

    Property FieldNames As Variant . . . . . 10-22

    Sub Clone(ppStruct As MWStruct) . . . . . 10-23

<b>Class MWField</b> .....	<b>10-25</b>
Property Name As String .....	<b>10-25</b>
Property Value As Variant .....	<b>10-25</b>
Property MWFlags As MWFlags .....	<b>10-25</b>
Sub Clone(ppField As MWField) .....	<b>10-25</b>
<b>Class MWComplex</b> .....	<b>10-27</b>
Property Real As Variant .....	<b>10-27</b>
Property Imag As Variant .....	<b>10-27</b>
Property MWFlags As MWFlags .....	<b>10-28</b>
Sub Clone(ppComplex As MWComplex) .....	<b>10-28</b>
<b>Class MWSparse</b> .....	<b>10-30</b>
Property NumRows As Long .....	<b>10-30</b>
Property NumColumns As Long .....	<b>10-30</b>
PropertyRowIndex As Variant .....	<b>10-30</b>
Property ColumnIndex As Variant .....	<b>10-31</b>
Property Array As Variant .....	<b>10-31</b>
Property MWFlags As MWFlags .....	<b>10-31</b>
Sub Clone(ppSparse As MWSparse) .....	<b>10-31</b>
<b>Class MWArg</b> .....	<b>10-34</b>
Property Value As Variant .....	<b>10-34</b>
Property MWFlags As MWFlags .....	<b>10-34</b>
Sub Clone(ppArg As MWArg) .....	<b>10-34</b>
<b>Enum mwArrayFormat</b> .....	<b>10-36</b>
<b>Enum mwDataType</b> .....	<b>10-37</b>
<b>Enum mwDateFormat</b> .....	<b>10-38</b>

**Data Conversion**

**A**

<b>Data Conversion Rules</b> .....	<b>A-2</b>
<b>Array Formatting Flags</b> .....	<b>A-11</b>
<b>Data Conversion Flags</b> .....	<b>A-13</b>
CoerceNumericToType .....	<b>A-13</b>
InputDateFormat .....	<b>A-14</b>
OutputAsDate As Boolean .....	<b>A-15</b>
DateBias As Long .....	<b>A-15</b>

**Troubleshooting**

**B**

<b>Errors and Solutions</b> .....	<b>B-2</b>
MATLAB Builder EX Errors and Suggested Solutions .....	<b>B-2</b>
Required Locations to Develop and Use Components .....	<b>B-4</b>
Microsoft Excel Errors and Suggested Solutions .....	<b>B-5</b>
Function Wizard Problems .....	<b>B-6</b>
<b>Deployment Product Terms</b> .....	<b>B-9</b>



# Getting Started

---

- “MATLAB Builder EX Product Description” on page 1-2
- “MATLAB Application Deployment Products ” on page 1-3
- “How MATLAB Builder EX Works” on page 1-5
- “MATLAB Builder EX Prerequisites” on page 1-6
- “Choosing Function Deployment Workflow” on page 1-10
- “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12
- “Integrate an Add-In and COM Component with Microsoft Excel” on page 1-16
- “Next Steps” on page 1-21

## **MATLAB Builder EX Product Description**

### **Deploy MATLAB code as Microsoft Excel add-ins**

MATLAB<sup>®</sup> Builder<sup>™</sup> EX lets you package MATLAB programs as Microsoft<sup>®</sup> Excel<sup>®</sup> add-ins and integrate them into Excel spreadsheets. These add-ins enable you to perform analyses and simulations within Excel that include MATLAB math, graphics, and user interfaces developed in MATLAB. The add-ins can be distributed royalty-free to computers that do not have MATLAB installed using the MATLAB Compiler Runtime (MCR) that is provided with MATLAB Compiler<sup>™</sup>.

Using MATLAB Compiler, MATLAB Builder EX encrypts your MATLAB programs and then generates Microsoft Visual Basic<sup>®</sup> for Applications (VBA) wrappers around them so that they behave just like other Excel add-ins.

MATLAB Builder EX creates VBA macros that let you pass Excel worksheet values to a deployed MATLAB program without writing or editing VBA code. It lets you prototype in Excel, debug your MATLAB source code, build the add-in, and load and test it in the worksheet.

### **Key Features**

- Royalty-free distribution of Excel add-ins to computers that do not have MATLAB installed
- Automatic conversion between Excel and MATLAB data types
- One-click VBA macro creation for interacting with a MATLAB application from within Excel
- Ability to prototype in Excel and debug source code in MATLAB

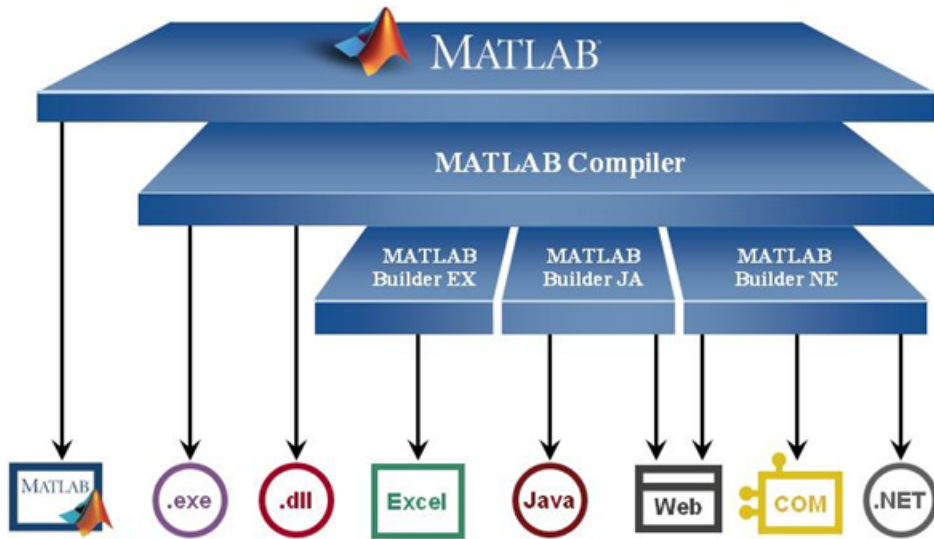


## MATLAB Application Deployment Products

The following table and figure summarizes the target applications supported by each product.

### MATLAB Suite of Application Deployment Products

Product	Target	Standalone Applications	Function Libraries	Graphical Apps	Web Apps	WebFigures
MATLAB Compiler	Standalone applications and C and C++ shared libraries	Yes	Yes	Yes	No	No
MATLAB Builder NE	C# .NET assemblies Visual Basic COM components	No	Yes	Yes	Yes	Yes
MATLAB Builder JA	Java <sup>®</sup> packages	No	Yes	Yes	Yes	Yes
MATLAB Builder EX	Microsoft Excel add-ins	No	Yes	Yes	No	No



### **MATLAB Application Deployment Products**

As this figure illustrates, each of the builder products uses the MATLAB Compiler core code to create deployable artifacts.

## How MATLAB Builder EX Works

The builder converts MATLAB functions to methods of a class that you define. From this class, the builder creates Excel add-ins (an `.xla` file, in addition to a `.bas` file).

MATLAB Builder EX components are Microsoft® COM objects that are accessible from Microsoft Excel through Visual Basic® for Applications (VBA). MATLAB Builder EX integrates the COM wrapper with the MATLAB Compiler-generated VBA code, saving you considerable development resources and time.

*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 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 project. The COM components created by MATLAB Builder EX contain a single class. This class provides the interface to the MATLAB functions that you add to the class at build time. The COM component provides a set of methods that wrap the MATLAB code and a DLL file.

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

For information about how MATLAB® Compiler works, see “How Does MATLAB Compiler Software Build My Application?” on page 5-2.

## MATLAB Builder EX Prerequisites

**In this section...**

- “Your Role in the Deployment Process” on page 1-6
- “Products, Compilers, and IDE Installation” on page 1-7
- “Macro Execution Security Levels in Microsoft Excel” on page 1-7
- “Deployment Target Architectures and Compatibility” on page 1-8
- “Dependency and Non-Compilable Code Considerations” on page 1-9
- “For More Information” on page 1-9

### Your Role in the Deployment Process

The table Application Deployment Roles, Goals, and Tasks describes the different roles, or jobs, that MATLAB Builder EX users typically perform. It also describes tasks they would most likely perform when running the examples in this documentation.

You may occupy one or more of the following roles.

#### Application Deployment Roles, Goals, and Tasks

Role	Knowledge Base	Responsibilities
MATLAB programmer	<ul style="list-style-type: none"> <li>• Understand the end-user business requirements and the mathematical models that support them.</li> <li>• MATLAB expert</li> <li>• No IT experience</li> </ul>	<ul style="list-style-type: none"> <li>• Build a Microsoft Excel add-in with MATLAB tools.</li> <li>• Package the component for distribution to customers.</li> <li>• Pass the package to the Microsoft Excel developer for further integration into the end-user environment.</li> </ul>
Microsoft Excel developer	<ul style="list-style-type: none"> <li>• Little or no MATLAB experience.</li> </ul>	<ul style="list-style-type: none"> <li>• Roll out the packaged component and integrate</li> </ul>

Role	Knowledge Base	Responsibilities
	<ul style="list-style-type: none"> <li>• Microsoft Excel expert.</li> <li>• Proficient writing VB/VBA code..</li> </ul>	<p>it into the end-user environment.</p> <ul style="list-style-type: none"> <li>• Write VB/VBA code to complement or augment the Excel Add-in built by the MATLAB programmer. Add and modify code as needed.</li> <li>• Uses Function Wizard to customize the add-in and create executable macros.</li> <li>• Verify that the final application executes reliably in the end-user environment.</li> </ul>

## Products, Compilers, and IDE Installation

Install the following products to run the example described in this chapter:

- MATLAB
- MATLAB Compiler
- MATLAB Builder EX
- A supported C or C++ compiler

## Macro Execution Security Levels in Microsoft Excel

If you will be creating macros and generating add-ins with MATLAB Builder EX, adjust the security settings accordingly in Microsoft Excel.

Failure to do so may result in add-ins not being generated or warning messages sent to MATLAB Compiler

Depending on what version of Microsoft Office you are using, do one of the following:

- For Microsoft Office 2003:

- 1 Click **Tools > Macro > Security**.
  - 2 For **Security Level**, select **Medium**.
- For Microsoft Office 2007:
    - 1 Click the 2007 Office ribbon button.
    - 2 Click **Excel Options > Trust Center > Trust Center Settings > Macro Settings**.
    - 3 In **Developer Macro Settings**, select **Trust access to the VBA project object model**.
  - For Microsoft Office 2010:
    - 1 Click **File > Options > Trust Center > Trust Center Settings > Macro Settings**.
    - 2 In **Developer Macro Settings**, select **Trust access to the VBA project object model**.

## Deployment Target Architectures and Compatibility

Before you deploy a component with MATLAB Builder EX, consider if your target machines are 32-bit or 64-bit.

Applications developed on one architecture must be compatible with the architecture on the system where they are deployed.

For example, if you have a 64-bit system, you usually install a 64-bit version of MATLAB (and most other applications), by default. Running functions you have developed with a 64-bit version of MATLAB requires a Function Wizard installed with a 64-bit version of Microsoft Excel.

### Migration Considerations for 32-bit and 64-bit Microsoft Excel

Add-ins created with MATLAB Builder are not compatible with versions of Microsoft Excel that do not share the same bitness:

- Add-ins created with 64-bit versions MATLAB Builder EX are *only* compatible with 64-bit versions of Microsoft Excel.
- Add-ins created with 32-bit versions MATLAB Builder EX are *only* compatible with 32-bit versions of Microsoft Excel.

## Dependency and Non-Compilable Code Considerations

Before you deploy your code, examine the code for dependencies on functions that may not be compatible with MATLAB Compiler.

For more detailed information about dependency analysis (`depfun`) and how MATLAB Compiler evaluates MATLAB code prior to compilation, see “Write Deployable MATLAB Code” in the MATLAB Compiler documentation.

### For More Information

If you...	See...
Want to verify your MATLAB code or function can be deployed successfully	“Write Deployable MATLAB Code”
Know your function is deployable and want to select a Getting Started workflow	“Choosing Function Deployment Workflow” on page 1-10

## Choosing Function Deployment Workflow

<b>In this section...</b>
“Is Your Function Ready for Deployment?” on page 1-10
“Other Examples” on page 1-10

### Is Your Function Ready for Deployment?

<b>If These Statements are True...</b>	<b>See...</b>
<ul style="list-style-type: none"> <li>• I have a MATLAB function that conforms to guidelines in “Write Deployable MATLAB Code”.</li> <li>• I want to create a Microsoft Excel compatible add-in from my existing MATLAB function.</li> <li>• I have tested and debugged my MATLAB function with MATLAB.</li> </ul>	<p>“Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12 to build your add-in and</p> <p>“Integrate an Add-In and COM Component with Microsoft Excel” on page 1-16 to execute the newly built add-in and create macros and associated GUIs with the Function Wizard and Microsoft Excel.</p>
<ul style="list-style-type: none"> <li>• I have not yet developed a MATLAB function for deployment as an add-in or I am in the process of developing it.</li> <li>• I have not tested my MATLAB function thoroughly with MATLAB.</li> </ul>	<p>“End-to-End Deployment of a MATLAB Function Using the Function Wizard” on page 4-22 for an end-to-end example of creating, debugging, building, and packaging MATLAB function from scratch using the Function Wizard.</p>

### Other Examples

For other types of examples, see the following:

- “Execute a Graphical Function” on page 6-11
- “Create Dialog Box and Error Message Macros” on page 6-15
- “Work with Variable-Length Inputs and Outputs”
- MATLAB Central. Set the Search field to **File Exchange** and search for one or more of the following:
  - InterpExcelDemo



- MatrixMathExcelDemo
- ExcelCurveFit

## Create a Microsoft Excel Add-In and COM Component from MATLAB Code

This example shows how to create a Microsoft Excel add-in. The example wraps a MATLAB function, `mymagic`, which computes a magic square. (A *magic square* is simply a square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.) You can use this example as a template for deploying functions with no output, scalar output, or multidimensional matrix output.

### Create an Excel Add-in in MATLAB

This example shows how to create an Excel Add-in using a MATLAB function. You can then hand the generated add-in off to an Excel developer who is responsible for integrating it into an application.

In MATLAB, examine the MATLAB code that want to deploy as a shared library.

- a Create the function that you want to deploy.

---

**Note:** MATLAB Builder EX includes this example code, called `mymagic.m`, in the `matlabroot\toolbox\matlabxl\examples\xlmagic` folder, where `matlabroot` represents the name of your MATLAB installation folder.

---

```
function y = mymagic(x)
%MYMAGIC Magic square of size x.
% Y = MYMAGIC(X) returns a magic square of size x.
% This file is used as an example for the MATLAB
% Builder EX product.

% Copyright 2001-2007 The MathWorks, Inc.
```

- b Run the example in MATLAB to make sure it runs.

```
mymagic(5)

17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
```

11 18 25 2 9

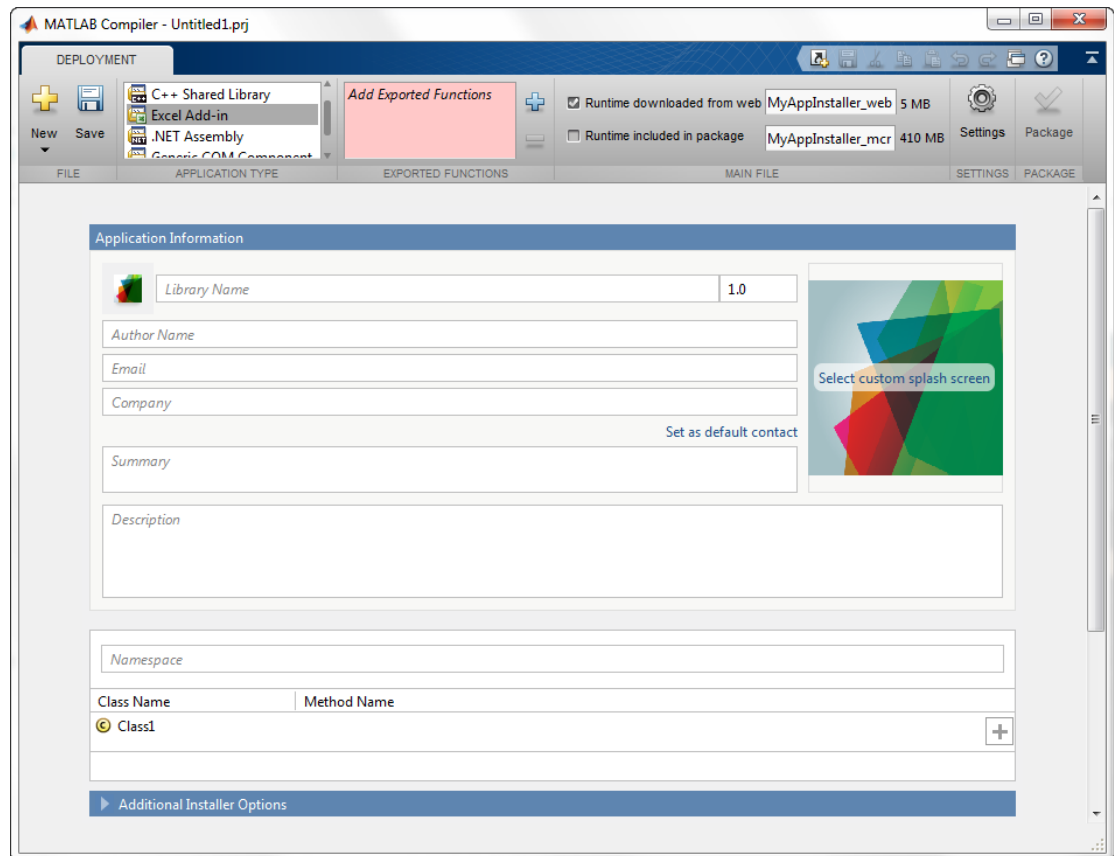
Open the **Library Compiler** app.

- a On the toolstrip, select the **Apps** tab.
- b Click the arrow at the far right of the tab to open the apps gallery.
- c Click **Library Compiler**.

---

**Note:** You can also call the `libraryCompiler` command.

---



In the **Application Type** section of the toolstrip, select **Excel Add-in** from the list.

---

**Note:** If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

---

Specify the MATLAB functions you want to deploy.

- a** In the **Exported Functions** section of the toolstrip, click the plus button.

---

**Note:** If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

---

- b** In the file explorer that opens, locate and select the `mymagic.m` file.

`mymagic.m` is located in `matlabroot\toolbox\matlabx1\examples\xlmagic`.

- c** Click **Open** to select the file and close the file explorer.

The Library Compiler adds the file you select (**mymagic.m**) to the list of files and a minus button appears under the plus button. The Library Compiler uses the name of the file as the name of the deployment project file (`.prj`), shown in the title bar, and as the name of the add-in, shown in the first field of the Application Information area. The project file saves all of the deployment settings so that you can re-open the project. The compiler uses the add-in name as the name of the component output, shown in the Application Information area, and as the name of the method in the class it creates.

In the **Packaging Options** section of the toolstrip, verify that the **Runtime downloaded from web** check box is selected.

---

**Note:** If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

---

This option creates an application installer that automatically downloads the MATLAB runtime and installs it along with the deployed add-in.

Explore the main body of the **MATLAB Compiler** project window.

The project window is divided into the following areas:

- **Application Information** — Editable information about the deployed application. This information is used by the generated installer to populate the installed application's metadata.
- **Additional Installer Options** — The default installation path for the generated installer.
- **Files required for your application** — Additional files required by the generated application. These files will be included in the generated application installer.
- **Files installed with your application** — Files that are installed with your application. These files include a Visual Basic file, a DLL, an Excel Add-in file, and a read me text file.
- **Additional Runtime Settings** — Access to other compiler options.

Click **Package**.

The Package window opens while the library is being generated.

The packaging process generates a self-extracting file that *automatically registers* the DLL and unpacks all deployable deliverables. MATLAB Builder EX first attempts to register the component to all users on the machine. This generally requires administrator privileges, however. If it cannot do this, it defaults to attempting to register under the current user name.

Select the **Open output folder when process completes** check box.

When the deployment process is complete, a file explorer opens and displays the generated output.

It should contain:

- `for_redistribution` — A folder containing the installer to distribute the library
- `for_testing` — A folder containing the raw files generated by the compiler
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the library
- `PackagingLog.txt` — A log file generated by the compiler

Click **Close** on the Package window.

## Integrate an Add-In and COM Component with Microsoft Excel

In this section...
“Files Necessary for Deployment” on page 1-16
“Add-in and COM Component Registration” on page 1-17
“COM Component Incorporation into Microsoft Excel using the Function Wizard” on page 1-18
“Distributing MATLAB Code Using the MATLAB Runtime” on page 1-18
“Add-in Installation and Distribution” on page 1-20

### Key Tasks for the Microsoft Excel End User

Task	Reference
Verify that you have received all necessary files from the MATLAB programmer.	“Files Necessary for Deployment” on page 1-16
Verify registry permissions for the add-in file and associated component.	“Add-in and COM Component Registration” on page 1-17
Execute your generated functions and create macros.	“Executing Functions and Creating Macros Using the Function Wizard” on page 4-2
Install the MATLAB runtime on target systems and update system paths.	“Distributing MATLAB Code Using the MATLAB Runtime” on page 1-18
Use the Excel add-in.	“Add-in Installation and Distribution” on page 1-20

### Files Necessary for Deployment

Before beginning, verify that you have access to the following files:

- The MCR Installer. For locations of all MCR Installers, run the mcrinstaller command.
- .xla file (the add-in)
- .bas file (the generated VBA code)
- .dll file

- `readme.txt`

## Add-in and COM Component Registration

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

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

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

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

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

---

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

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

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

- 1 Unregister the component with this command:  
`mwregsvr /u /useronly my_dll.dll`
  - 2 Reregister the component to the `LOCAL MACHINE` hive with this command:
-

---

```
mwregsvr my_dll.dll
```

---

## COM Component Incorporation into Microsoft Excel using the Function Wizard

Now that your add-in and COM component have been created, use the Function Wizard to integrate the COM component into Microsoft Excel.

See “Executing Functions and Creating Macros Using the Function Wizard” on page 4-2 for a complete example of how to execute functions and create macros using the Magic Square example in this chapter.

## Distributing MATLAB Code Using the MATLAB Runtime

On target computers without MATLAB, install the MATLAB runtime, if it is not already present on the deployment machine.

### Install MATLAB Runtime

The *MATLAB runtime* is an execution engine made up of the same shared libraries MATLAB uses to enable execution of MATLAB files on systems without an installed version of MATLAB.

The “MATLAB runtime” is now available for downloading from the Web to simplify the distribution of your applications created using the MATLAB Compiler. Download the MATLAB runtime from the MATLAB runtime product page.

The MATLAB runtime installer does the following:

- 1 Installs the MATLAB runtime (if not already installed on the target machine)
- 2 Installs the component assembly in the folder from which the installer is run
- 3 Copies the `MWArray` assembly to the Global Assembly Cache (GAC), as part of installing the MATLAB runtime

### MATLAB Runtime Prerequisites

- 1 Since installing the MATLAB runtime requires write access to the system registry, ensure you have administrator privileges to run the MATLAB Runtime Installer.



- 2 The version of the MATLAB runtime that runs your application on the target computer must be compatible with the version of MATLAB Compiler that built the deployed MATLAB code.
- 3 Do not install the MATLAB runtime in MATLAB installation directories.
- 4 The MATLAB runtime installer requires approximately 2 GB of disk space.

### Add the MATLAB Runtime Installer to the Installer

This example shows how to include the MATLAB runtime in the generated installer, using one of the compiler apps. The generated installer contains all files needed to run the standalone application or shared library built with MATLAB Compiler and properly lays them out on a target system.

- 1 On the **Packaging Options** section of the compiler interface, select one or both of the following options:
  - **Runtime downloaded from web** — This option builds an installer that invokes the MATLAB runtime installer from the MathWorks Web site.
  - **Runtime included in package** — The option includes the MATLAB runtime installer into the generated installer.
- 2 Click **Package**.
- 3 Distribute the installer as needed.

### Install the MATLAB Runtime

This example shows how to install the MATLAB runtime on a system.

If you are given an installer containing the compiled artifacts, then the MATLAB runtime is installed along with the application or shared library. If you are given just the raw binary files, download the MATLAB runtime installer from the Web and run the installer.

---

**Note:** If you are running on a platform other than Windows, “set the system paths” on the target machine. Setting the paths enables your application to find the MATLAB runtime.

Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See “Using MATLAB Compiler on Mac or Linux” for detailed information on performing all deployment tasks specifically with UNIX variants such as Linux and Mac.

---

## Add-in Installation and Distribution

Since Microsoft Excel add-ins are written directly to the `distrib` folder by MATLAB Builder EX, you and your end users install them exactly as you installed the Function Wizard in “Installation of the Function Wizard” on page 4-4.

### Calling Add-in Code from the Excel Spreadsheet

To run the executable code from a cell in the Excel spreadsheet, invoke the add-in name with a method call. For example, if you deployed a piece of MATLAB code called `mymagic.m`, or a figure called `mymagic.fig`, you invoke that code by entering the following in a cell in the spreadsheet:

```
=mymagic()
```

---

**Tip** If the method call does not evaluate immediately, press **Ctrl**, **Shift**, and **Enter** simultaneously.

---

## Next Steps

MATLAB Builder EX includes several examples, in addition to the magic square example. You can find these examples in folders in `matlabroot\toolbox\matlabxl\examples\`. The following table identifies examples by folder:

For Example Files Relating To...	Find Example Code in Folder...	For Example Documentation See...
Magic Square Example	<code>xlmagic</code>	“Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12 “Integrate an Add-In and COM Component with Microsoft Excel” on page 1-16
Variable-Length Argument Example	<code>xlmulti</code>	“Work with Variable-Length Inputs and Outputs” on page 6-5
Calling Compiled MATLAB Functions from Microsoft Excel	<code>xlbasic</code>	“Calling Compiled MATLAB Functions from Microsoft Excel” on page 8-4
Spectral Analysis Example	<code>xlspectral</code>	“Build and Integrate with Spectral Analysis” on page 7-16

The following topics detail some of the more common tasks you perform as you continue to develop your applications.

To:	See:
Try more examples using MATLAB Builder EX	MATLAB Central. Set the Search field to <code>File Exchange</code> and search for one or more of the following: <ul style="list-style-type: none"> <li>• <code>InterpExcelDemo</code></li> <li>• <code>MatrixMathExcelDemo</code></li> <li>• <code>ExcelCurveFit</code></li> </ul>

<b>To:</b>	<b>See:</b>
Learn how to write MATLAB code that is optimized for deployability	“Write Deployable MATLAB Code”
Work with functions having graphical output	“Execute a Graphical Function” on page 6-11
Work with functions having variable inputs and output	“Work with Variable-Length Inputs and Outputs”
Create displayable dialog boxes and error messages	“Create Dialog Box and Error Message Macros” on page 6-15
Troubleshoot common error messages	Appendix B
Integrate your application into your enterprise environment by enhancing your application's generated Visual Basic code	“Code Your Microsoft Visual Basic Application” on page 7-3

# Customizing a Compiler Project

---

- “Customize the Installer” on page 2-2
- “Manage Required Files in a Compiler Project” on page 2-6
- “Specify Files to Install with the Application” on page 2-8
- “Manage Support Packages” on page 2-9

# Customize the Installer

### In this section...

- “Change the Application Icon” on page 2-2
- “Add Application Information” on page 2-3
- “Change the Splash Screen” on page 2-3
- “Change the Installation Path” on page 2-4
- “Change the Logo” on page 2-4
- “Edit the Installation Notes” on page 2-5

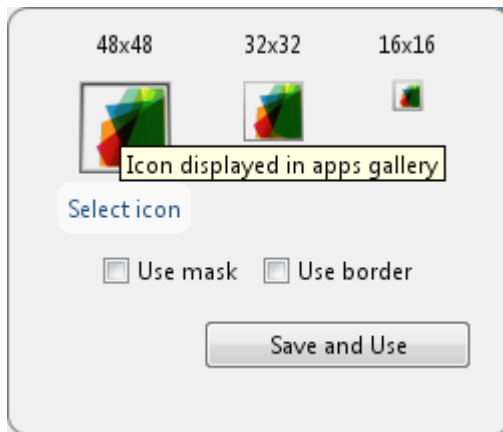
## Change the Application Icon

The application icon is used for the generated installer. For standalone applications, it is also the application's icon.

You can change the default icon in **Application Information**. To set a custom icon:

- 1 Click the graphic to the left of the **Application name** field.

A window previewing the icon opens.



- 2 Click **Select icon**.
- 3 Using the file explorer, locate the graphic file to use as the application icon.
- 4 Select the graphic file.

- 5 Click **OK** to return to the icon preview.
- 6 Select **Use mask** to fill any blank spaces around the icon with white.
- 7 Select **Use border** to add a border around the icon.
- 8 Click **Save and Use** to return to the main compiler window.

## Add Application Information

The **Application Information** section of the compiler app allows you to provide these values:

- Name

Determines the name of the installed MATLAB artifacts. For example, if the name is `foo`, the installed executable would be `foo.exe`, the Windows start menu entry would be `foo`. The folder created for the application would be `InstallRoot/foo`.

The default value is the name of the first function listed in the **Main File(s)** field of the compiler.

- Version

The default value is 1.0.

- Author name
- Support e-mail address
- Company name

Determines the full installation path for the installed MATLAB artifacts. For example, if the company name is `bar`, the full installation path would be `InstallRoot/bar/ApplicationName`.

- Summary
- Description

This information is all optional and, unless otherwise stated, is only used for display purposes. It appears on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

## Change the Splash Screen

The installer's splash screen displays after the installer is started. It is displayed, along with a status bar, while the installer initializes.

You can change the default image by clicking the **Select custom splash screen** link in **Application Information**. When the file explorer opens, locate and select a new image.

---

**Note:** You can drag and drop a custom image onto the default splash screen.

---

## Change the Installation Path

Default Installation Paths lists the default path the installer will use when installing the compiled binaries onto a target system.

### Default Installation Paths

Windows	C:\Program Files \ <i>companyName</i> \ <i>appName</i>
Mac OS X	/Applications/ <i>companyName</i> / <i>appName</i>
Linux®	/usr/ <i>companyName</i> / <i>appName</i>

You can change the default installation path by editing the **Default installation folder** field under **Additional Installer Options**.

The **Default installation folder** field has two parts:

- root folder — A drop down list that offers options for where the install folder is installed. Custom Installation Roots lists the optional root folders for each platform.

### Custom Installation Roots

Windows	C:\Users\ <i>userName</i> \AppData
Linux	/usr/local

- install folder — A text field specifying the path appended to the root folder.

## Change the Logo

The logo displays after the installer is started. It is displayed on the right side of the installer.

You change the default image by clicking the **Select custom logo** link in **Additional Installer Options**. When the file explorer opens, locate and select a new image.



---

**Note:** You can drag and drop a custom image onto the default logo.

---

## Edit the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. They can provide useful information concerning any additional set up that is required to use the installed binaries or simply provide instructions for how to run the application.

The field for editing the installation notes is in **Additional Installer Options**.

# Manage Required Files in a Compiler Project

In this section...
“Dependency Analysis” on page 2-6
“Using the Compiler Apps” on page 2-6
“Using mcc” on page 2-6

## Dependency Analysis

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to compile and run. These files are automatically compiled into the generated binary. The compiler does not generate any wrapper code allowing direct access to the functions defined by the required files.

## Using the Compiler Apps

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required by your application to run** field.

To add files:

- 1 Click the plus button in the field.
- 2 Select the desired file from the file explorer.
- 3 Click **OK**.

To remove files:

- 1 Select the desired file.
- 2 Press the **Delete** key.

---

**Caution** Removing files from the list of required files may cause your application to not compile or to not run properly when deployed.

---

## Using mcc

If you are using `mcc` to compile your MATLAB code, the compiler does not display a list of required files before running. Instead, it compiles all of the required files that are

discovered by the dependency analysis function and adds them to the generated binary file.

You can add files to the list by passing one, or more, `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all of the files in `foo`, and its subfolders, to the list of required files.

### Specify Files to Install with the Application

The compiler apps package files to install along with the ones it generates. By default the installer includes a readme file with instructions on installing the MATLAB Compiler Runtime and configuring it.

These files are listed in the **Files installed with your application** section of the app.

to add files to the list:

- 1 Click the plus button in the field.
- 2 Select the desired file from the file explorer.
- 3 Click **OK** to close the file explorer.

To remove files from the list:

- 1 Select the desired file.
- 2 Press the **Delete** key.

---

**Caution** Removing the binary targets from the list results in an installer that does not install the intended functionality.

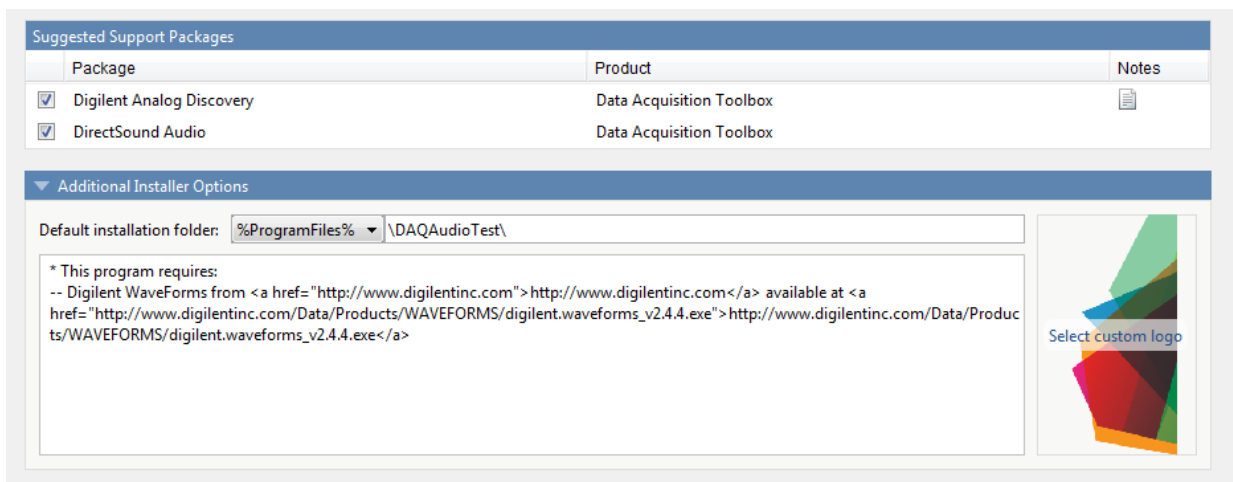
---

When installed on a target computer, the files listed in the **Files installed with your application** are placed in the **application** folder.

## Manage Support Packages

### Using a Compiler App

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, the compiler app displays a **Suggested Support Packages** section.



The list displays all installed support packages that your MATLAB code requires. The list is determined using these criteria:

- the support package is installed
- your code has a direct dependency on the support package
- your code is dependent on the base product of the support package
- your code is dependent on at least one of the files listed as a dependency in the `mcc.xml` file of the support package, and the base product of the support package is MATLAB

Deselect support packages that are not required by your application.

Some support packages require third-party drivers that the compiler cannot package. In this case, the compiler adds the information to the installation notes. You can edit

installation notes in the **Additional Installer Options** section of the app. To remove the installation note text, deselect the support package with the third-party dependency.

---

**Caution** Any text you enter beneath the generated text will be lost if you deselect the support package.

---

### Using the Command Line

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, you pass a `-a` flag to `mcc` when compiling your MATLAB code.

For example, if your function uses the `OS Generic Video Interface` support package.

```
mcc -m -v test.m -a C:\MATLAB\SupportPackages\R2014a\genericvideo
```

Some support packages require third-party drivers that the compiler cannot package. In this case, you are responsible for downloading and installing the required drivers.

# Integrating with MATLAB Production Server

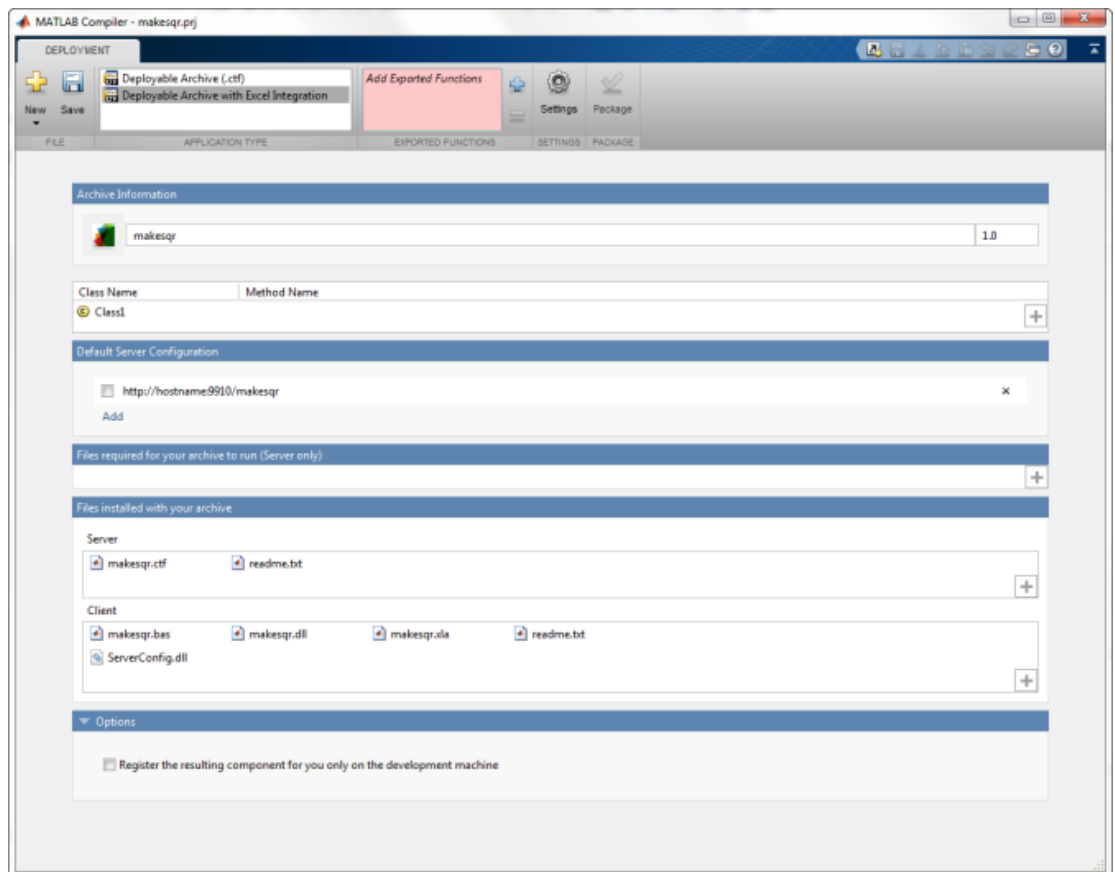
---

- “Build Deployable Archive and Add-In” on page 3-2
- “Install the Deployable Archive” on page 3-5
- “Install the Excel Add-In” on page 3-6
- “Data Marshaling Rules” on page 3-8
- “Change the Server Configuration” on page 3-10
- “Enable Client Logging” on page 3-12
- “XLA File Not Generated” on page 3-14
- “Server Configuration Add-in Not Enabled” on page 3-15
- “Error in myfunc: Object reference not set to an instance of an object” on page 3-16

## Build Deployable Archive and Add-In

To create an Excel add-In that integrates with MATLAB Production Server™:

- 1 Open the **Production Server Compiler**.
  - a On the toolstrip, select the **Apps** tab.
  - b Click the arrow at the far right of the tab to open the apps gallery.
  - c Click **Production Server Compiler** to open the project window.



- 2 In the **Application Type** section of the toolstrip, select **Deployable Archive with Excel Integration** from the list.
- 3 Specify the MATLAB functions you want to deploy.



- a** In the **Exported Functions** section of the toolstrip, click the plus button.
- b** In the file explorer that opens, locate and select the desired files.
- c** Click **Open** to select the files and close the file explorer.

The selected files are added to the list of files and a minus button appears under the plus button.

---

**Note:** Functions that return a variable number of outputs are not supported by add-ins that use code running on a MATLAB Production Server instance.

---

- 4** Inspect the **Archive Information** section of the app.

The first text field is the name of the archive. The name of the archive determines the names of the generated artifacts and the URL used to connect to the server.

- 5** Inspect the class mapping table to ensure that all desired functions are being compiled.
- 6** If you need to change the marshaling rules for a function, select **Data Conversion Properties** from the function name's context menu.

For more information, see “Data Marshaling Rules” on page 3-8.

- 7** Optionally configure the default server configuration packaged with the installer.

The server configuration defines the connection to the MATLAB Production Server instance running the MATLAB code.

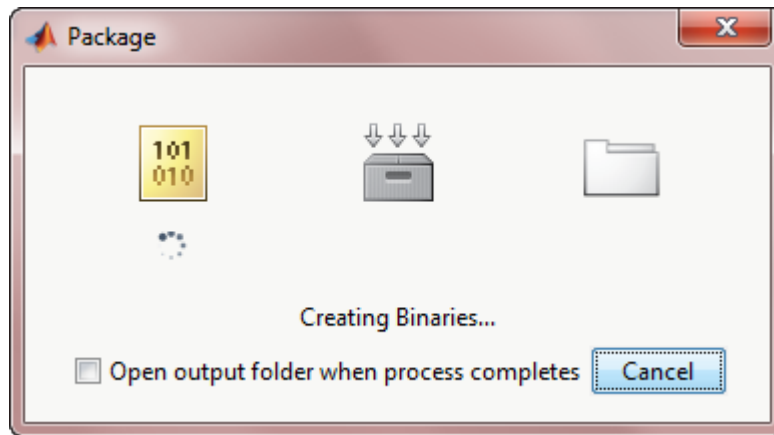
- a** Search the **Default Server Configuration** table for the URL to package with the installer.
  - b** If it is in the table, select it.
  - c** If not, click **Add** to add it to the table.
- 8** Inspect the **Files required for your archive to run** and **Files installed with your archive** sections of the app.

These sections of the app list all of the files that are packaged with the compiled code.

**Files required for your archive to run** lists the files on which your function is dependent. They are packaged into the deployable archive and are only for the server. See “Manage Required Files in a Compiler Project” on page 2-6.

**Files installed with your archive** includes sections for both the client and the server. The files listed are generated by the compiler and should be delivered to the person installing the application.

- 9 Click **Package** to generate the add-in and the deployable archive.



- 10 Select the **Open output folder when process completes** check box to display the generated output.

When the deployment process is complete, a file explorer opens and displays the generated output.

- 11 Click **Close** on the Package window.

- 12 Verify the contents of the generated output:

- `for_redistribution` — A `client` folder containing the generated installer and a `server` folder containing a `.zip` file
- `for_testing` — A `client` folder containing the raw files generated for the add-in and a `server` folder containing the raw files generated for the deployable archive
- `for_redistribution_files_only` — A `client` folder containing only the files needed to redistribute the add-in and a `server` folder containing only the files needed to redistribute the deployable archive
- `PackagingLog.txt` — A log file generated by the compiler

## Install the Deployable Archive

The Production Server Compiler generates a deployable archive containing the compiled MATLAB code used by the add-in. This archive needs to be deployed to a MATLAB Production Server instance before the add-in will work.

To install the deployable archive on a server instance:

- 1 Locate the archive in the `for_redistribution_files_only\server\` folder.

The file name is similar to `archiveName.ctf`.

- 2 Copy the archive file to the `auto_deploy` folder of the server instance.

For more information, see “MATLAB Production Server” documentation.

## Install the Excel Add-In

### In this section...

“Install with the Generated Installer” on page 3-6

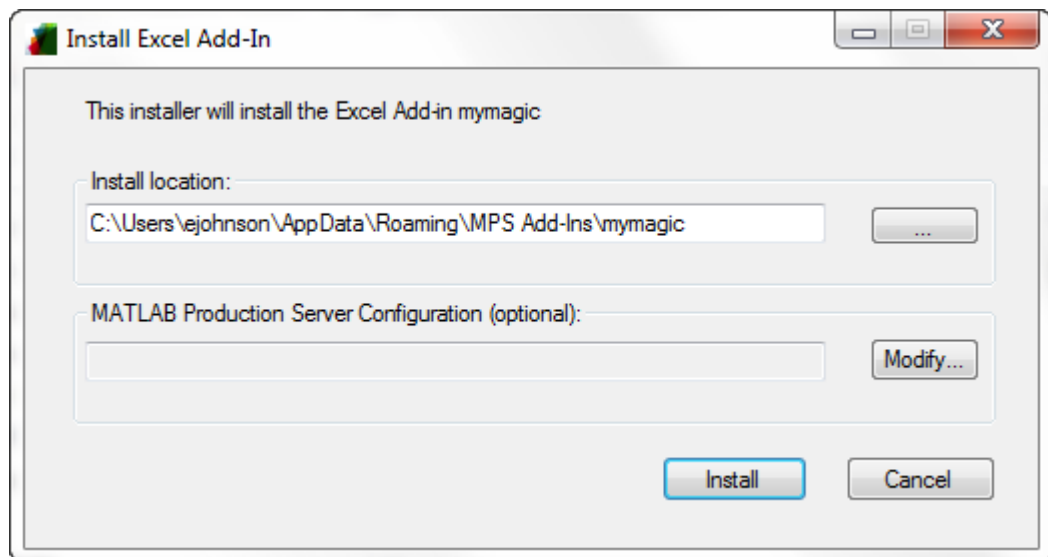
“Install with the Raw Files” on page 3-7

### Install with the Generated Installer

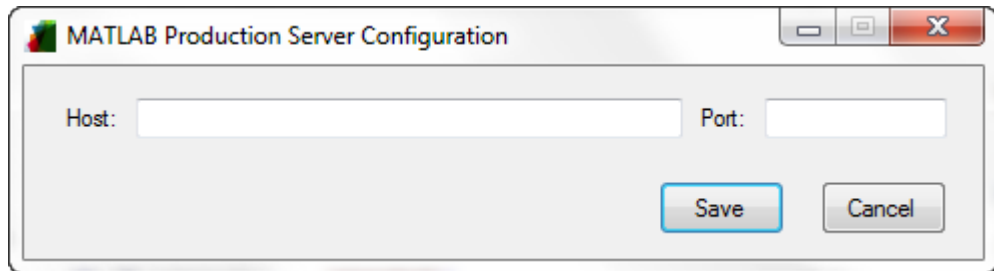
The Production Server Compiler generates an installer that you use to install the add-in and configure it to connect to a MATLAB Production Server instance.

To install the add-in:

- 1 Locate the installer in the `for_redistribution\client` folder.
- 2 Double click the installer to start it.



- 3 In the **Install location** field, enter the location where the Excel add-in is installed.
- 4 Specify the URL of the deployable archive powering the Excel add-in.
  - a Click **Modify**.



- b** In the **Host** field, enter the host name or the full URL of the machine running the MATLAB Production Server instance with the deployable archive.
  - c** In the **Port** field, enter the port number on which the server instance listens for incoming requests.  
 The default port number is 9910.
  - d** Click **Save**.
- 5** Click **Install**.

## Install with the Raw Files

In the `client\for_redistribution_files_only` you will find the following files:

- `projName.dll`— Dynamic library required by the add-in
- `projName.bas` — VBA module file created for the add-in
- `projName.xla` — Excel add-in file

---

**Note:** If `projName.xla` is not generated, you can still install the add-in using `projName.bas`. See “XLA File Not Generated” on page 3-14 for more information.

---

- `serverconfig.dll`- Excel add-in that manages the server configuration

On the machine running the add-in, you must register the following DLLs with the `regAsm /codebase` command:

- `mymagic.dll`
- `serverconfig.dll`

If `mymagic.xla` is generated, install the add-in into Excel. If not, import `mymagic.bas` to the workbook's Visual Basic project.

## Data Marshaling Rules

<b>In this section...</b>
“Default Marshaling Rules” on page 3-8
“Change Rules for Marshaling Data into MATLAB” on page 3-8
“Change Rules for Marshaling Data into Excel” on page 3-9

### Default Marshaling Rules

These types of data do not have natural mappings between MATLAB and Excel:

- Dates: Excel has a special data type for dates, and MATLAB does not.
- Blank cells: MATLAB has no equivalent construct for a blank cell in an Excel spread sheet.

If you do not change the marshaling rules when compiling the add-in, the rules for marshaling Excel data into MATLAB are:

- Excel dates are marshaled into MATLAB doubles.
- Empty cells are marshaled into zeros.

If you do not change the marshaling rules when compiling the add-in, the rules for marshaling MATLAB data into Excel are:

- MATLAB NaNs are marshaled into Visual Basic #QNANs.
- MATLAB does not return any Excel dates.

### Change Rules for Marshaling Data into MATLAB

You can change how dates and empty cells are marshaled into MATLAB when compiling the add-in:

- Excel dates can be marshaled as MATLAB strings.
- Empty cells can be marshaled as MATLAB NaNs.

To change the marshaling rules:

- 1 In the class mapper portion of the **MATLAB Compiler** project window, select the signature of the function you want to modify.

- 2 Select **Data Conversion Properties** from the context menu.
- 3 Select the input argument rules to change.
- 4 Click outside of the dialog box to close it.

## Change Rules for Marshaling Data into Excel

You can change how dates and NaNs are marshaled into Excel when compiling the add-in:

- MATLAB NaNs can be converted into zeros.
- MATLAB numeric values can be converted into Excel dates.

---

**Note:** To see a date in the expected format, ensure that the Excel cell is formatted to display its contents in a date format.

---

To change the marshaling rules:

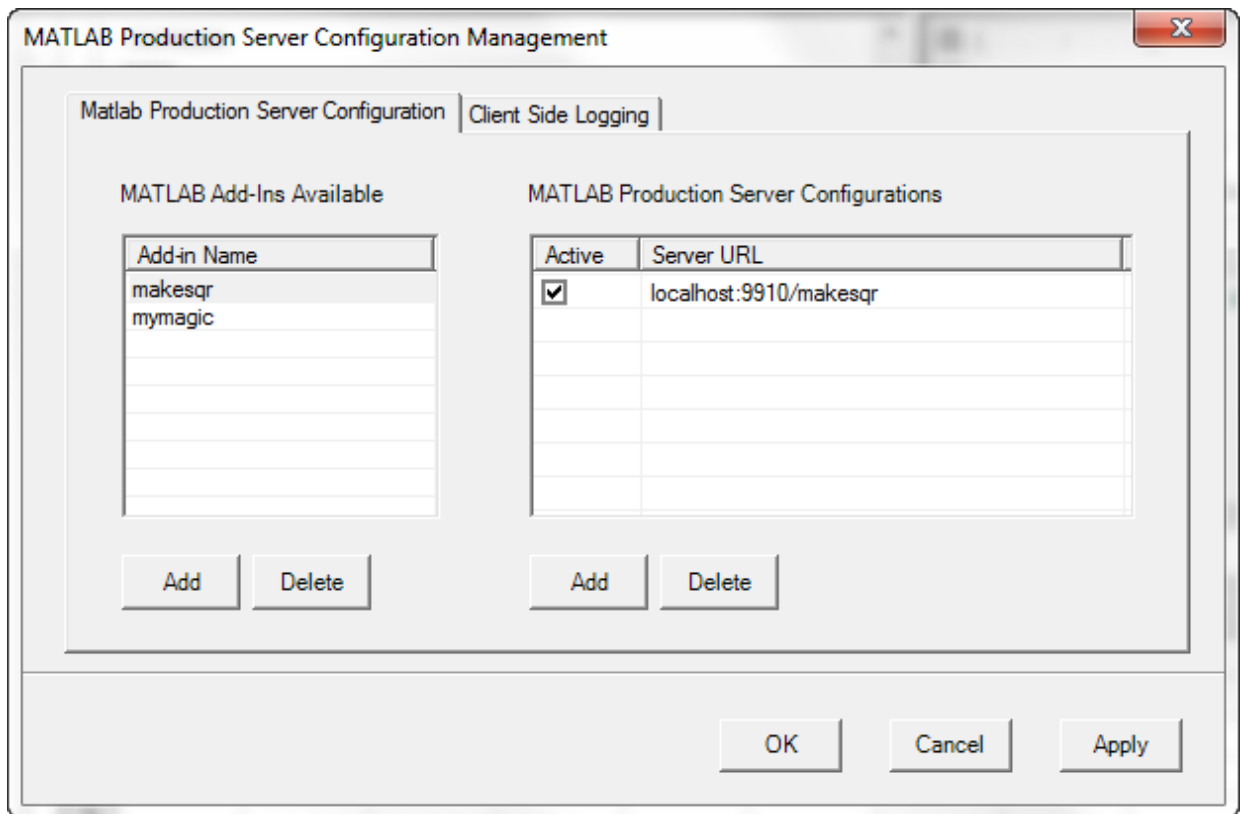
- 1 In the class mapper portion of the **MATLAB Compiler** project window, select the signature of the function you want to modify.
- 2 Select **Data Conversion Properties** from the context menu.
- 3 Select the output argument rules to change.
- 4 Click outside of the dialog to close it.

## Change the Server Configuration

Before you can use an add-in uses code running on a MATLAB Production Server instance, you must configure it to connect to a MATLAB Production Server instance.

To configure the server information for an add-in:

- 1 In Excel, select **Add-Ins > Configure MATLAB Production Server**.



- 2 Select the **MATLAB Production Server Configuration** tab.
- 3 Select the add-in to configure.

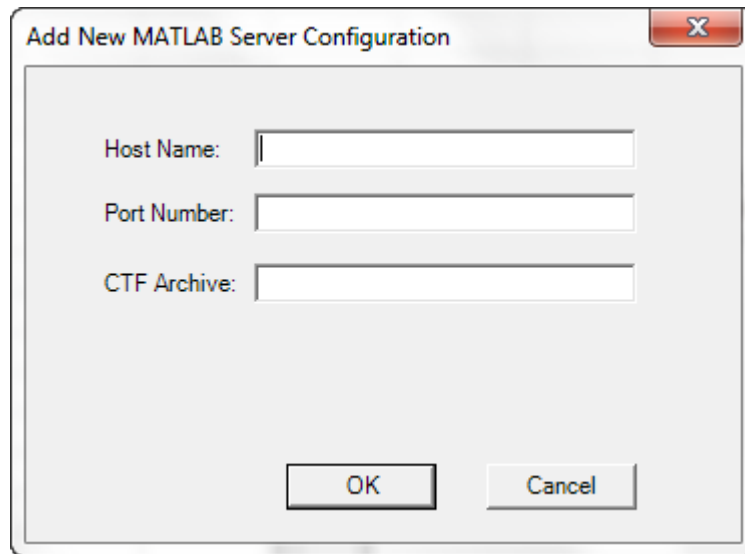
If the add-in is not listed, click the **Add** button under the **MATLAB Add-ins Available** table. Enter the name of the add-in and click **OK**.



- 4 Select the URL of the server hosting the deployable archive implementing the MATLAB function for the add-in.

If the server URL is not listed:

- 1 Click the **Add** button under the **MATLAB Production Server Configurations** table.



The image shows a dialog box titled "Add New MATLAB Server Configuration". It has a standard Windows-style title bar with a close button (X) in the top right corner. The main area of the dialog contains three text input fields, each with a label to its left: "Host Name:", "Port Number:", and "CTF Archive:". Below these fields are two buttons: "OK" and "Cancel".

- 2 Enter the host name for the server instance in the **Host Name** field.
- 3 Enter the port number on which the server instance listens for requests in the **Port Number** field.

The default port number is **9910**.

- 4 Enter the name of the deployable archive in the **CTF Archive** field.

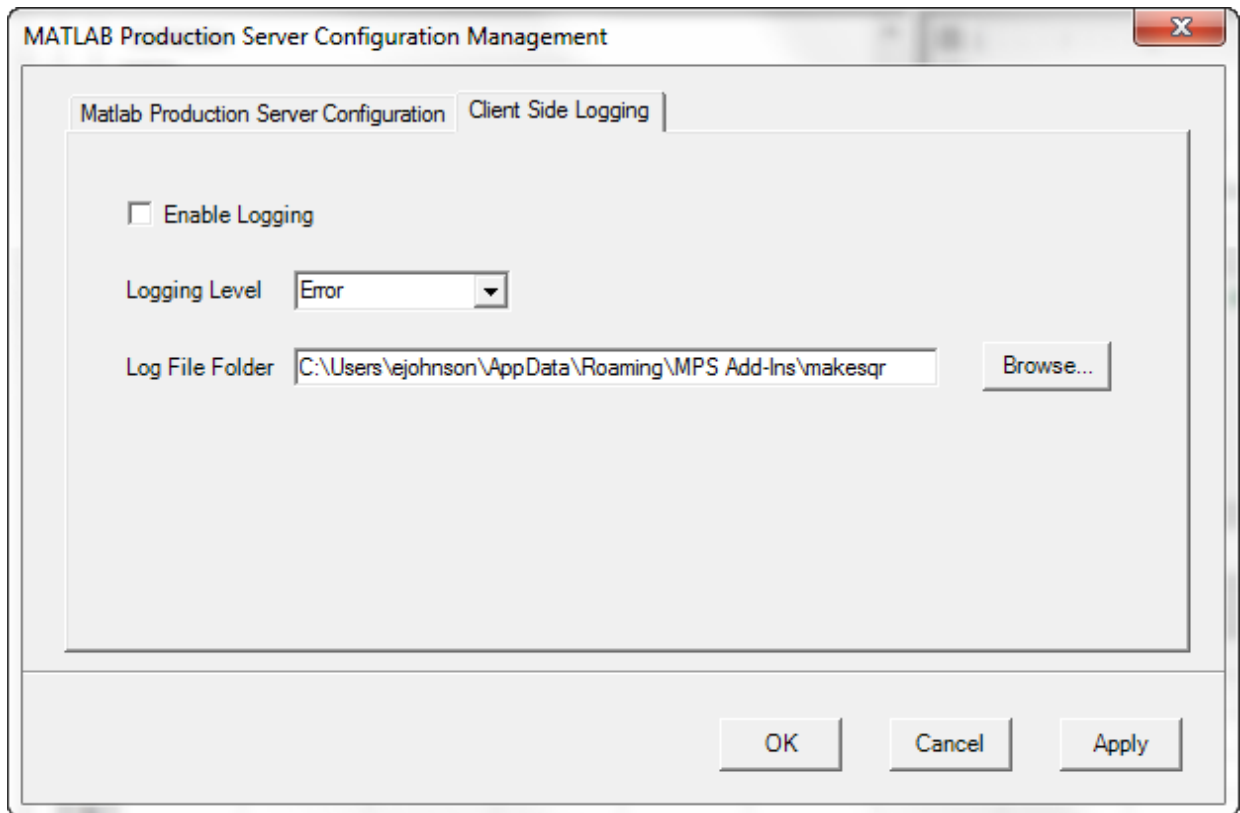
The name of the deployable archive is the archive name specified in the Production Server Compiler.

- 5 Click **OK**.
- 5 Click **OK**.

## Enable Client Logging

Client logging writes messages to a persistent log file. The messages are useful for debugging issues with the add-in. Client logging is disabled by default. To activate it:

- 1 In Excel, select **Add-Ins > Configure MATLAB Production Server**.



- 2 Select the **Client Side Logging** tab.
- 3 Check the **Enable Logging** box.
- 4 Select the logging level from the **Logging Level** drop down list.
  - **Error** — Log error messages.
  - **Information** — Log information about server interactions in addition to error messages.

- **Verbose** — Log everything.
- 5** In the **Log File Folder** box, enter the folder into which log files are written.

When client logging is enabled all registered MATLAB Production Server add-ins generate log files. The files follow the naming convention *logFolder\addinName-date.log*.

### **XLA File Not Generated**

The compiler may not generate the *projName.xla* file for a number of reasons, including that Excel is not configured to trust access to the VBA project object model. When this happens, you can install the add-in by importing the *projName.bas* file into the workbook's Visual Basic project.

## Server Configuration Add-in Not Enabled

If your trust settings in Excel are configured to either disable all add-ins or to require add-ins to be published by a trusted publisher, it is possible that the **Configure MATLAB Production Server** add-in is not available after installation. In most cases, the add-in is installed but disabled.

To check if the add-in is installed in Excel:

- 1 Select **File>Options**.
- 2 Select **Add-Ins**.
- 3 Look for **ServerConfig.Connect** in the list of disabled add-ins.

You can enable the add-in by adjusting the trust settings in Excel.

## **Error in myfunc: Object reference not set to an instance of an object**

If your add-in throws the error:

```
Error in myfunc: Object reference not set to an instance of an object
```

The likely cause is that the MATLAB function used by the add-in returns a variable number of outputs.

Add-ins using code run on a MATLAB Production Server instance do not support MATLAB functions that return a variable number of outputs. You can either rewrite your MATLAB function to return a fixed number of outputs, or you can create an add-in that runs locally to your Excel installation.

# The Function Wizard

---

- “Executing Functions and Creating Macros Using the Function Wizard” on page 4-2
- “End-to-End Deployment of a MATLAB Function Using the Function Wizard” on page 4-22

## Executing Functions and Creating Macros Using the Function Wizard

### In this section...

- “What Can the Function Wizard Do for Me?” on page 4-3
- “Installation of the Function Wizard” on page 4-4
- “Function Wizard Start-Up” on page 4-5
- “Workflow Selection for MATLAB Functions Ready for Deployment” on page 4-6
- “Defining Functions Ready to Execute” on page 4-7
- “Function Execution” on page 4-17
- “Macro Creation” on page 4-18
- “Macro Execution” on page 4-18
- “Microsoft Visual Basic Code Access (Optional Advanced Task)” on page 4-19
- “For More Information” on page 4-21

If your MATLAB function is ready to be deployed and you have already built your add-in and COM component with the Deployment Tool, follow this workflow to incorporate your built COM component into Microsoft Excel using the Function Wizard. To follow the workflow in this section effectively, you must run “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12.

The Function Wizard also allows you to iteratively test, develop, and debug your MATLAB function. Using this end-to-end workflow assumes you are still be in the process of developing your MATLAB function for deployment. See “End-to-End Deployment of a MATLAB Function Using the Function Wizard” on page 4-22 for complete instructions for this workflow.

See “Choosing Function Deployment Workflow” on page 1-10 for further details.

### Key Tasks for the End User

Task	Reference
1. Install the Function Wizard.	“Installation of the Function Wizard” on page 4-4
2. Start the Function Wizard.	“Function Wizard Start-Up” on page 4-5



Task	Reference
3. Select the option to incorporate your built COM component into Microsoft Excel.	“Workflow Selection for MATLAB Functions Ready for Deployment” on page 4-6
4. Define the new MATLAB function you want to prototype by adding it to the Function Wizard and establishing input and output ranges.	“Defining Functions Ready to Execute” on page 4-7
5. Test your MATLAB function by executing it with the Function Wizard.	“Function Execution” on page 4-17
6. Create a macro.	“Macro Creation” on page 4-18
7. Execute the macro you created using the Function Wizard.	“Macro Execution” on page 4-18
8. Optionally inspect or modify the Microsoft Visual Basic code you generated with the COM component. Optionally, attach the macro you created to a GUI button.	“Microsoft Visual Basic Code Access (Optional Advanced Task)” on page 4-19

## What Can the Function Wizard Do for Me?

The Function Wizard enables you to pass Microsoft Excel (Excel 2000 or later) worksheet values to a compiled MATLAB model and then return model output to a cell or range of cells in the worksheet.

The Function Wizard provides an intuitive interface to Excel worksheets. You do not need previous knowledge of Microsoft Visual Basic for Applications (VBA) programming.

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

---

**Note:** The Function Wizard does not currently support the MATLAB `sparse`, and `complex` data types.



---

### Installation of the Function Wizard

Before you can use the Function Wizard, you must first install it as an add-in that is accessible from Microsoft Excel.

After you install the Function Wizard, the entry **MATLAB Functions** appears as an available Microsoft Excel add-in button.

#### Using Office 2007

- 1 Start Microsoft Excel if it is not already running.
- 2  Click the Office Button (  ) and select **Excel Options**.
- 3 In the left pane of the Excel Options dialog box, click **Add-Ins**.
- 4 In the right pane of the Excel Options dialog box, select **Excel Add-ins** from the **Manage** drop-down box.
- 5 Click **Go**.
- 6 Click **Browse**. Navigate to `matlabroot\toolbox\matlabxl\matlabxl\arch` and select `FunctionWizard2007.xlam`. Click **OK**.
- 7 In the Excel Add-Ins dialog box, verify that the entry **MATLAB Builder EX Function Wizard** is selected. Click **OK**.

#### Using Office 2010

- 1 Click the **File** tab.
- 2 On the left navigation pane, select **Options**.
- 3 In the Excel Options dialog box, on the left navigation pane, select **Add-Ins**.
- 4 In the Manage drop-down, select **Excel Add-Ins**, and click **Go**.
- 5 In the Add-Ins dialog box, click **Browse**.
- 6 Browse to `matlabroot/toolbox/matlabxl/matlabxl/arch`, and select `FunctionWizard2007.xlam`. Click **OK**.
- 7 In the Excel Add-Ins dialog, verify that the entry **MATLAB Builder EX Function Wizard** is selected. Click **OK**.

The Home tab of the Microsoft Office Ribbon should now contain the Function Wizard tile. See The Home Tab of the Microsoft Office Ribbon with Function Wizard Installed.

### Using Office 2003

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

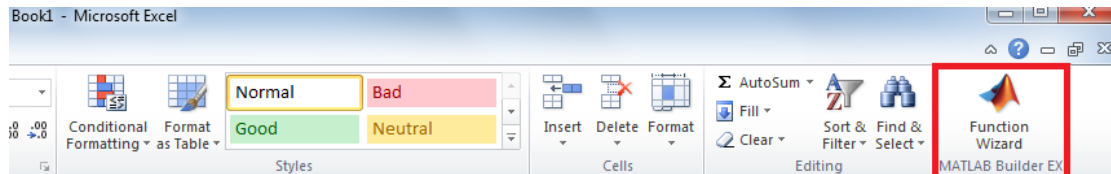
If the Function Wizard was not previously installed, click **Browse** and navigate to *matlabroot\toolbox\matlabxl\matlabxl* folder. Select *FunctionWizard.xla*. Click **OK** to proceed.

### Function Wizard Start-Up

Start the Function Wizard in one of the following ways. When the wizard has initialized, the Function Wizard Start Page dialog box displays.

#### Using Office 2007 or Office 2010

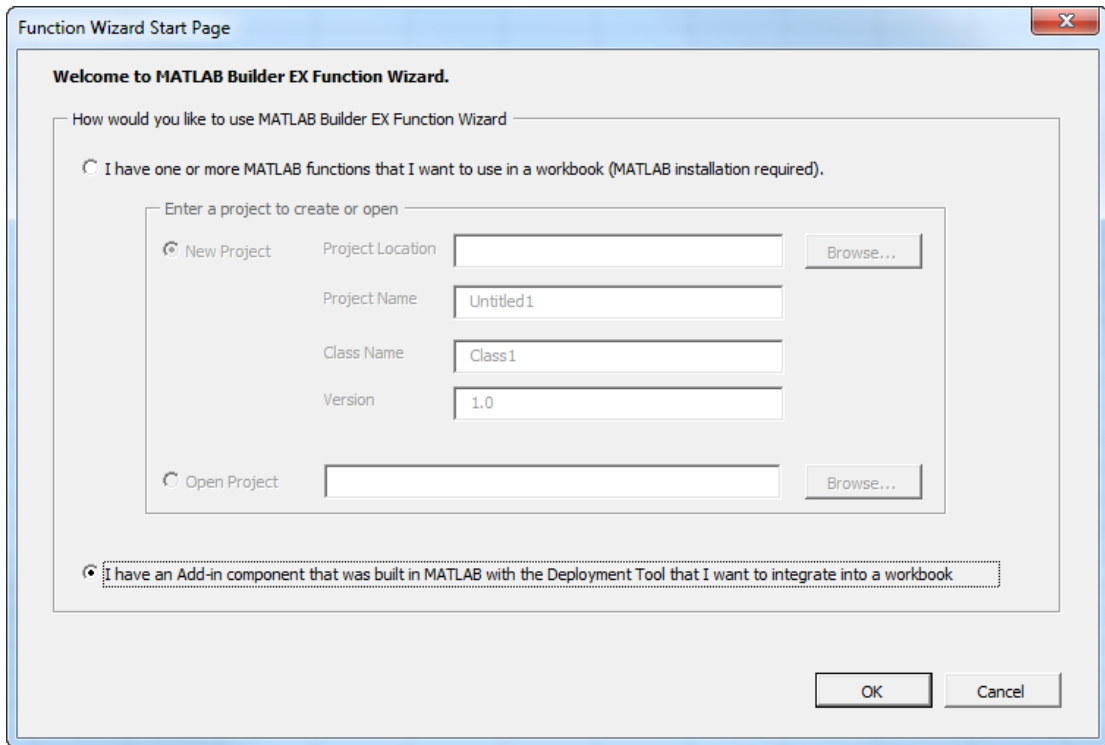
In Microsoft Excel, on the Microsoft Office ribbon, on the **Home** tab, select **MATLAB Functions**.



#### The Home Tab of the Microsoft Office Ribbon with Function Wizard Installed

##### Using Office 2003

- 1 Select **Tools > Add-Ins** from the Excel main menu.
- 2 Select **Function Wizard**.

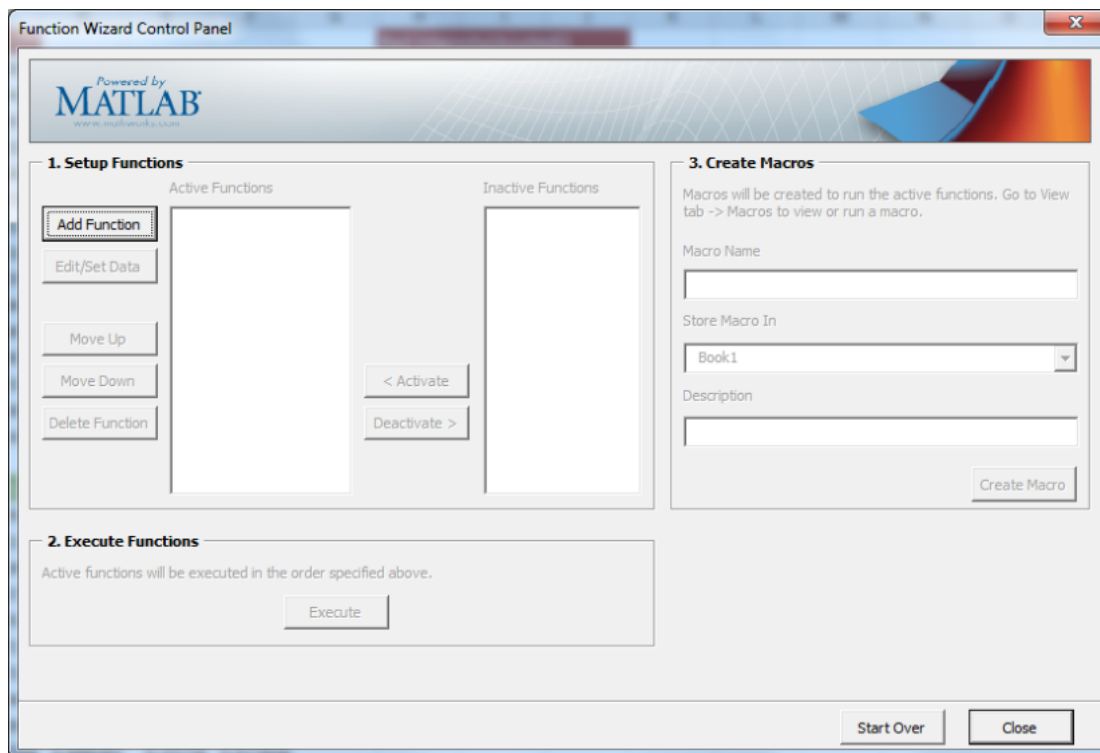


**The Function Wizard Start Page Dialog Box**

### **Workflow Selection for MATLAB Functions Ready for Deployment**

After you have installed and started the Function Wizard, do the following:

- 1 From the Function Wizard Start Page, select the option **I have an Add-in component that was built in MATLAB with the Deployment Tool that I want to integrate into a workbook.**
- 2 Click **OK**. The Function Wizard Control Panel opens with the **Add Function** button enabled..



## The Function Wizard Control Panel for Working with MATLAB Functions Ready for Deployment

---

**Tip** To return to the Function Wizard Start Page, click **Start Over**.

---

### Defining Functions Ready to Execute

- 1 Define the function you want to execute to the Function Wizard. Click **Add Function** in the Set Up Functions area of the Function Wizard Control Panel. The MATLAB Components dialog box opens.
- 2 In the Available Components area of the MATLAB Components dialog box, select the name of your component (xlmagic) from the drop-down box.

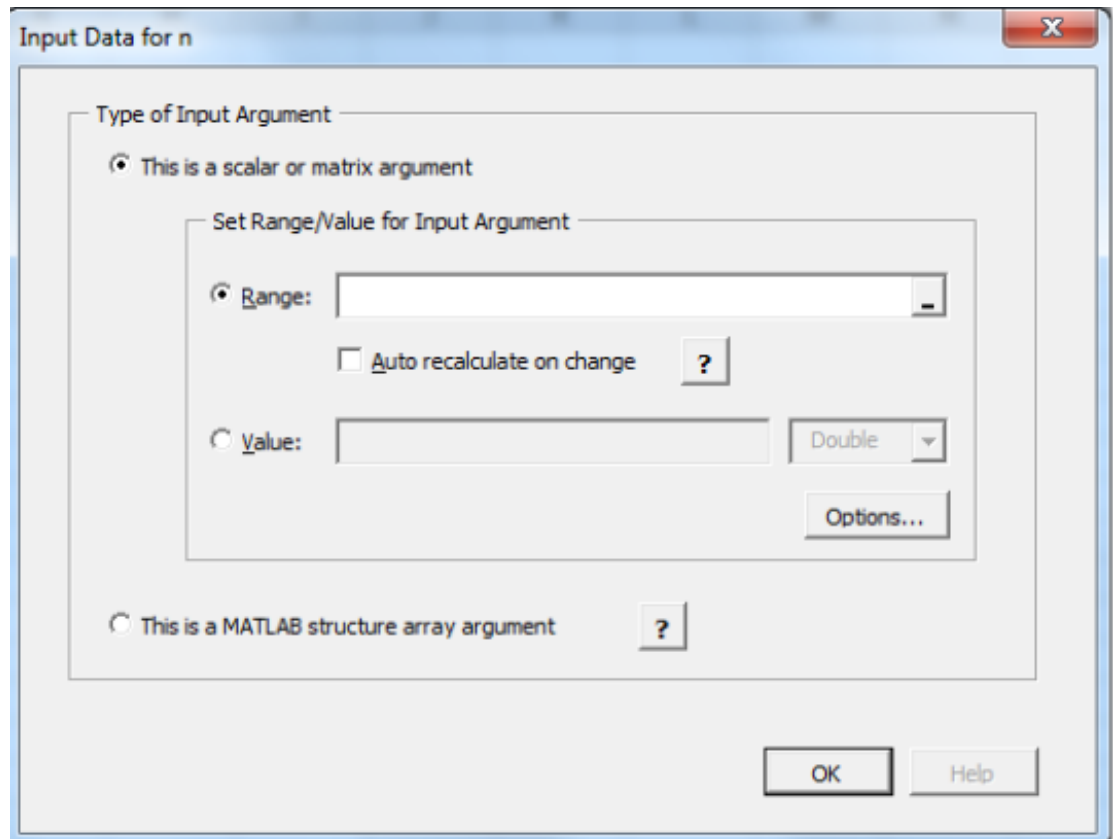
- 3 Select the function you want to execute (`mymagic`) from the box labeled **Functions for Class `xlmagic`**.
- 4 Click Add Function. The Function Properties dialog box opens.

---

**Tip** The **Function Syntax and Help** area, in the Function Properties dialog box, displays the first help text line (sometimes called the *H1 line*) in a MATLAB function. Displaying these comments in the Function Properties dialog box can be helpful when deploying new or unfamiliar MATLAB functions to end-users.

---

- 5 Define input argument properties as follows.
  - a On the **Input** tab, click **Set Input Data**. The Input Data for  $n$  dialog box opens.



- b** Specify a **Range** or **Value** by selecting the appropriate option and entering the value. If the argument refers to a *structure array* (*struct*), select the option **This is a MATLAB structure array argument**. See “Working with Struct Arrays” on page 4-12 for information on assigning ranges and values to fields in a struct array.

---

**Caution** Avoid selecting ranges using arrow keys. If you must use arrow keys to select ranges, apply the necessary fix from the Microsoft site: <http://support.microsoft.com/kb/291110>.

---

---

**Note:** Select the **Auto Recalculate on Change** option to force Microsoft Excel to automatically recalculate the spreadsheet data each time it changes.

---

- c** Click **OK**.

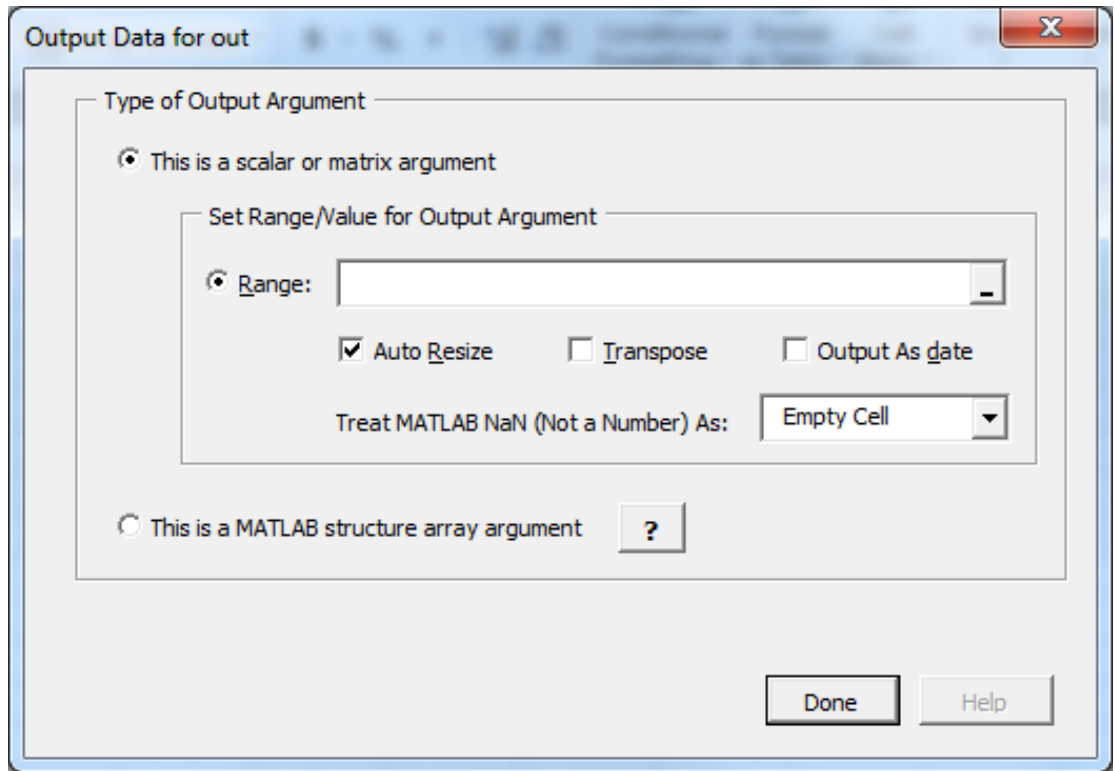
---

**Tip** To specify how MATLAB Builder EX handles blank cells (or cells containing no data), see “Empty Cell Value Control” on page 4-11.

---

- 6** Define output argument properties as follows.

- a** On the **Output** tab, click **Set Output Data**. The Output Data for *y* dialog box appears, where *y* is the name of the output variable you are defining properties of.



---

**Tip** You can also specify MATLAB Builder EX to **Auto Resize**, **Transpose** or output your data in date format (**Output as date**). To do so, select the appropriate option in the Argument Properties for  $y$  dialog box.

---

- b** Specify a **Range**. Alternately, select a range of cells on your Excel sheet; the range will be entered for you in the **Range** field.

---

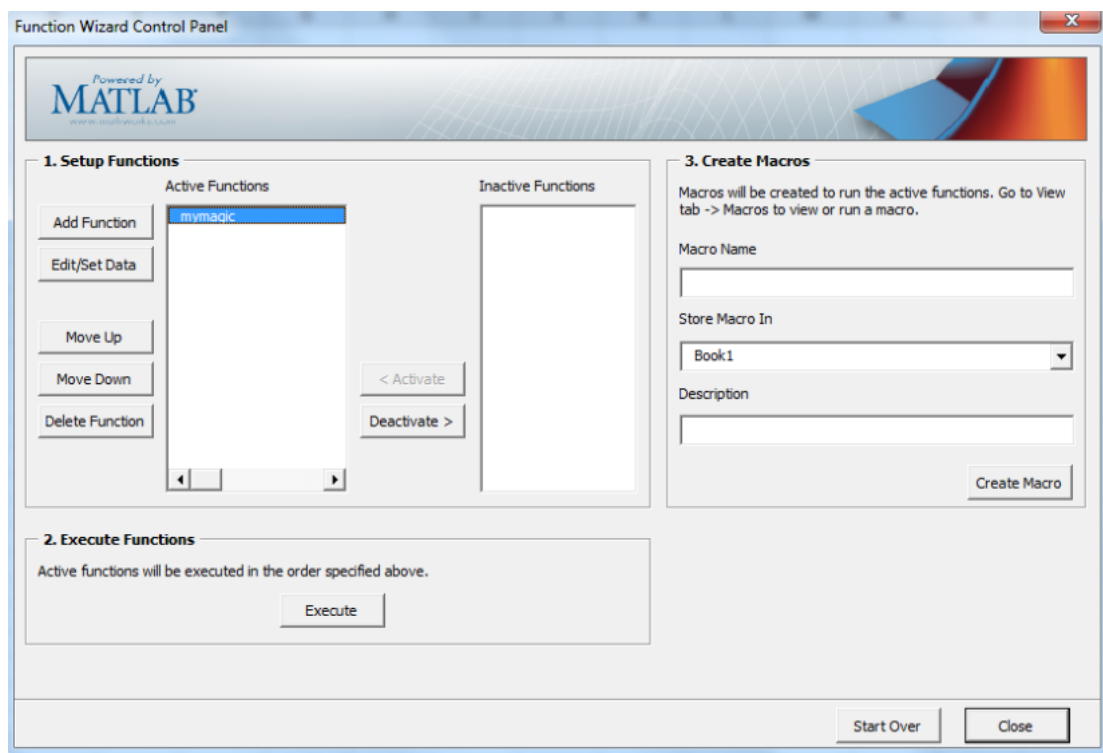
**Caution** Avoid selecting ranges using arrow keys. If you must use arrow keys to select ranges, apply the necessary fix from the Microsoft site: <http://support.microsoft.com/kb/291110>.

---

- c** Select **Auto Resize** if it is not already selected.
- d** Click **Done** in the Output Data for  $y$  dialog box.



- e Click **Done** in the Function Properties dialog box.



mymagic now appears in the **Active Functions** list of the Function Wizard Control Panel.

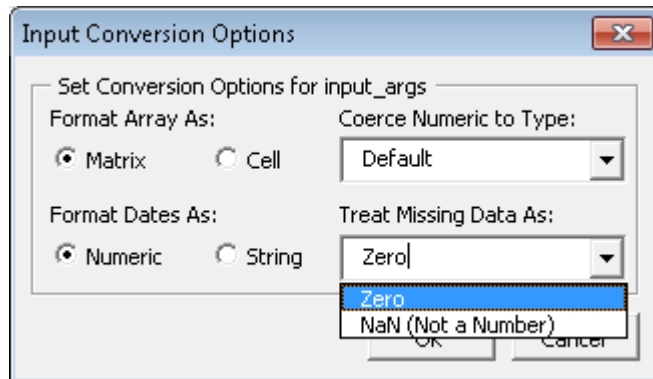
### Empty Cell Value Control

You can specify how MATLAB Builder EX processes empty cells, allowing you to assign undefined or unrepresented (NaN, for example) data values to them.

To specify how to handle empty cells, do the following.

- 1 Click **Options** in the Input Data for *N* dialog box.

The Input Conversion Options dialog box opens.

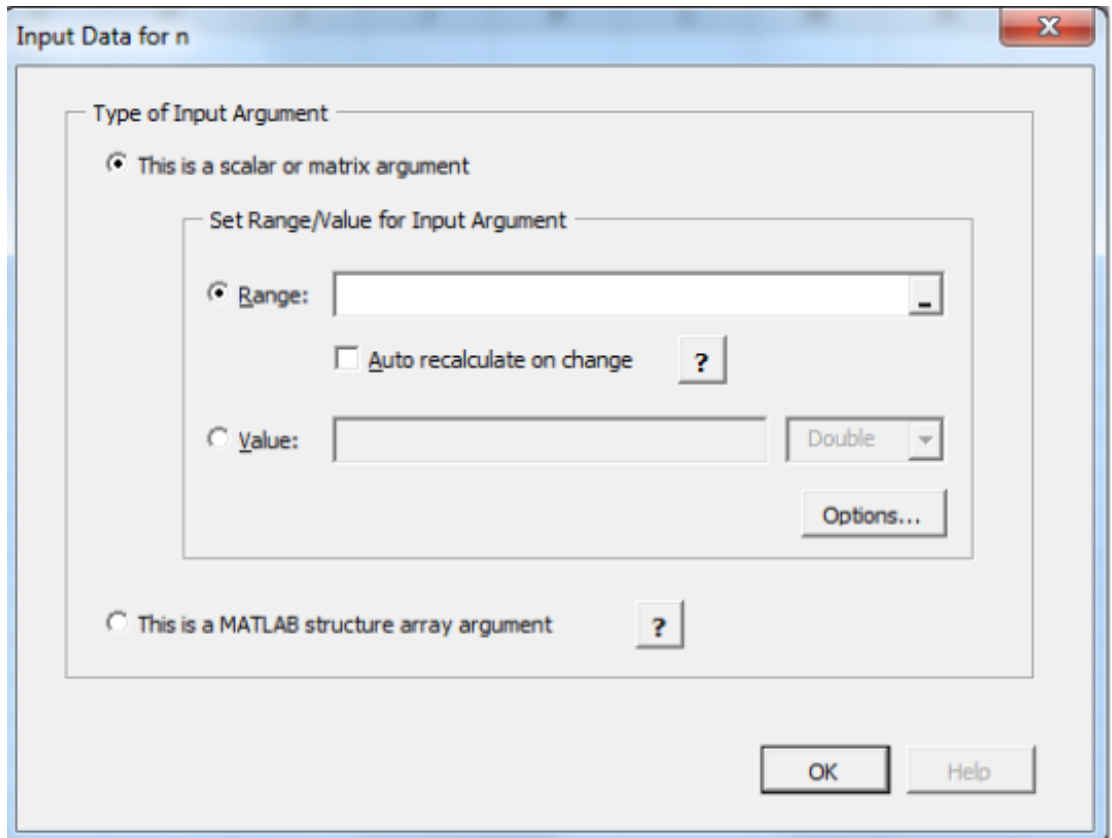


- 2 Click the **Treat Missing Data As** drop-down box.
- 3 Specify either **Zero** or **NaN (Not a Number)**, depending on how you want to handle empty cells.

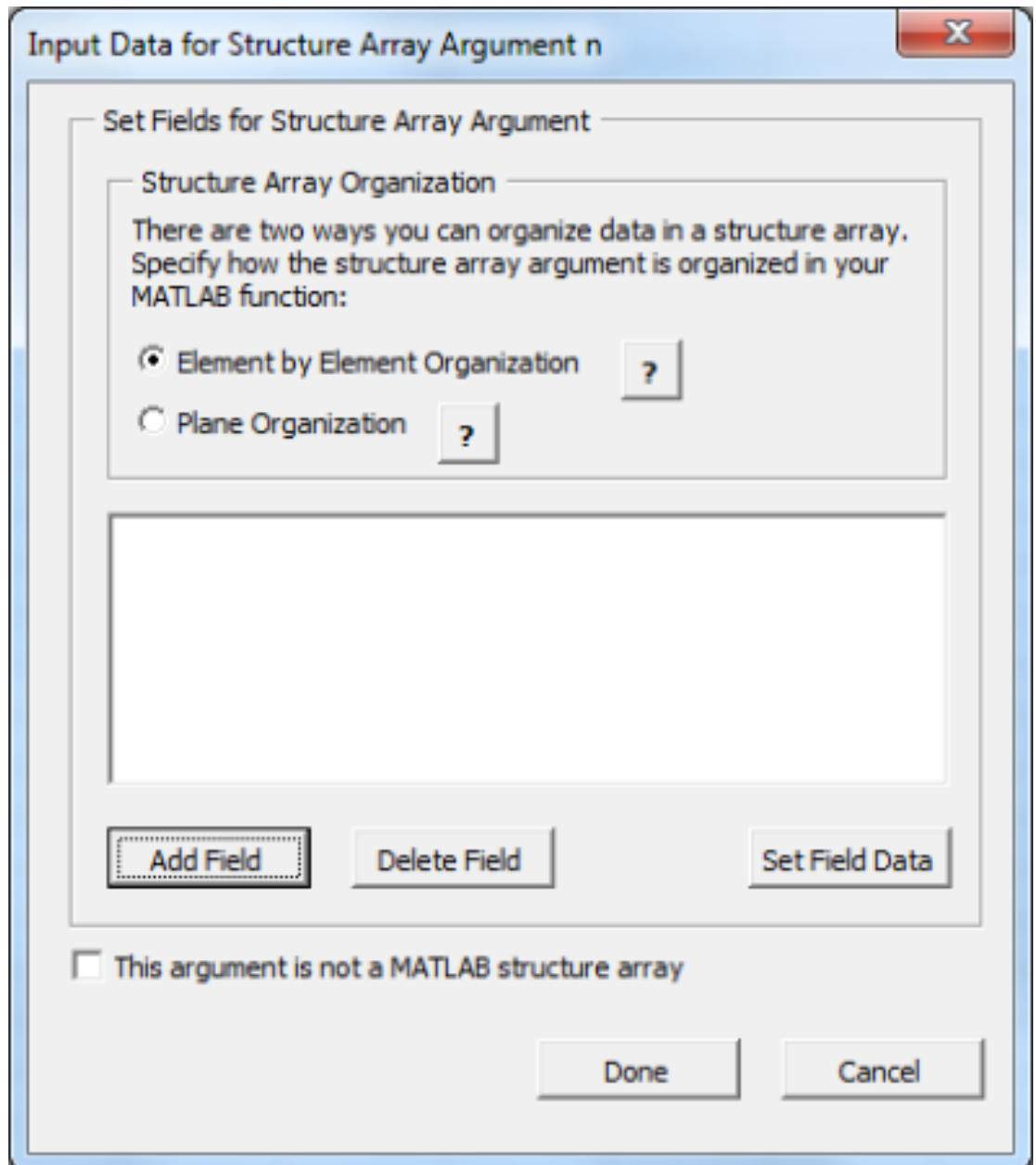
### Working with Struct Arrays

To assign ranges to fields in a struct array, do the following:

- 1 If you have not already done so, select **This is a MATLAB structure array argument** in the Input Data for  $n$  dialog box and click **OK**.



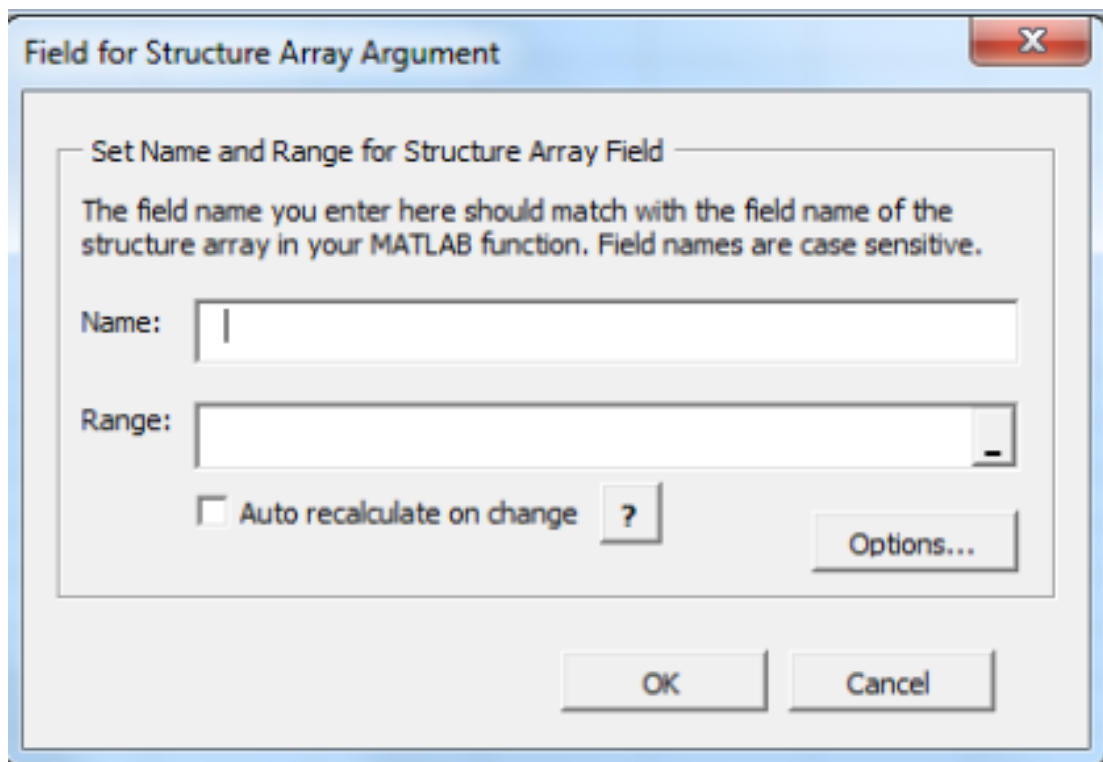
The Input Data for Structure Array Argument  $n$  dialog box opens.



- 2 The Function Wizard supports Vector and Two-dimensional struct arrays organized in either Element by Element or Plane organization, for both input and output.

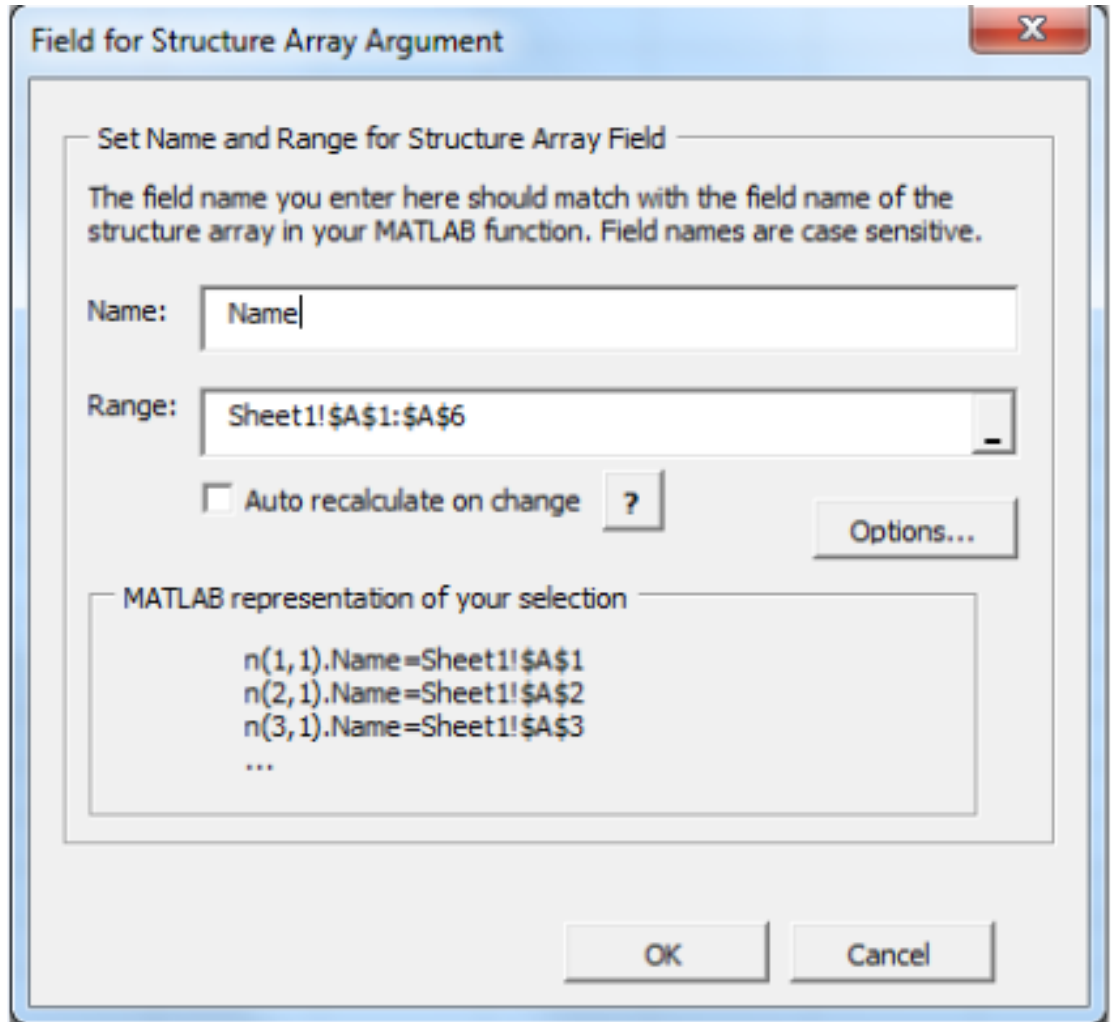
In the Input Data for Structure Array Argument  $n$  dialog box, do the following:

- a In the Structure Array Organization area, select either **Element by Element Organization** or **Plane Organization**.
- b Click **Add Field** to add fields for each of your struct array arguments. The Field for Structure Array Argument dialog box opens.



- 3 In the Field for Argument dialog box, do the following:
  - a In the **Name** field, define the field name. The **Name** you specify must match the field name of the structure array in your MATLAB function.
  - b Specify the **Range** for the field.

**Caution** Avoid selecting ranges using arrow keys. If you must use arrow keys to select ranges, apply the necessary fix from the Microsoft site: <http://support.microsoft.com/kb/291110>.



- c Click **Done**.

### How Structure Arrays are Supported

MATLAB Builder EX supports one and two-dimensional MATLAB structure arrays.

The product converts data passed into structure arrays in *element-by-element organization* or *plane organization*. See *MATLAB Programming Fundamentals* for more information about all MATLAB data types, including structures.

### Deploying Structure Arrays as Inputs and Outputs

If you are a MATLAB programmer and want to deploy a MATLAB function with structure arrays as input or output arguments, build Microsoft Excel macros using the Function Wizard and pass them (with the Excel Add-In and COM component) to the end users. If you can't do this, let your end users know:

- Which arguments are structure arrays
- Field names of the structure arrays

### Using Macros with Struct Arrays

The macro generation feature of MATLAB Builder EX works with struct arrays as input or output arguments. See “Macro Creation” on page 4-18 if you have a MATLAB function you are ready to deploy. See “Microsoft Excel Add-in and Macro Creation Using the Function Wizard” on page 4-44 if you are using the Function Wizard to create your MATLAB function from scratch. See “Choosing Function Deployment Workflow” on page 1-10 for more information on both workflows.

## Function Execution

In the Execute Functions area of the Function Wizard Control Panel, click **Execute** to run `mymagic`. Cells A1:E5 on the Excel sheet are automatically populated with the output of `mymagic` (a magic square of dimension 5).

A	B	C	D	E
17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

### Macro Creation

Continuing the example, create a Microsoft Excel macro using the Function Wizard Control Panel:

- 1 In the Create Macros area of the control panel, enter `mymagic` in the **Macro Name** field.
- 2 Select the location of where to store the macro in the **Store Macro** drop-down box.
- 3 Enter a brief description of the macro's functionality in the **Description** field.
- 4 Click **Create Macro**.

A macro is created in the current Excel workbook.

### Macro Execution

Run the macro you created in “Macro Creation” on page 4-18 by doing one of the following, after first clearing cells A1 : E5 (which contain the output of the Magic Square function you ran in “Function Execution” on page 4-17).

---

**Tip** You may need to enable the proper security settings before running macros in Microsoft Excel. For information about macro permissions and error messages occurring from executing macros, see the Appendix B appendix.

---

### Using Office 2007 or Office 2010

- 1 In Microsoft Excel, click **View > Macros > View Macros**.
- 2 Select `mymagic` from the **Macro name** drop-down box.



- 3 Click **Run**. Cells A1:E5 on the Excel sheet are automatically populated with the output of `mymagic`.

### Using Office 2003

- 1 In Microsoft Excel, click **Tools > Macro > Macros**.
- 2 Select `mymagic` from the **Macro name** drop-down box.
- 3 Click **Run**. Cells A1:E5 on the Excel sheet are automatically populated with the output of `mymagic`.

## Watch a Video

Watch a video that demonstrates how to execute deployed functions with the Function Wizard and Microsoft Excel.

### Microsoft Visual Basic Code Access (Optional Advanced Task)

Optionally, you may want to access the Visual Basic code or modify it, depending on your programming expertise or the availability of an Excel developer. If so, follow these steps.

From the Excel main window, open the Microsoft Visual Basic editor by doing one of the following. select **Tools > Macro > Visual Basic Editor**.

#### Using Office 2007 or Office 2010

- 1 Click **Developer > Visual Basic**.
- 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 `Matlab Macros` module.

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

#### Using Office 2003

- 1 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 `Matlab Macros` module.

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

### Mapping a Macro to a GUI Button or Control (Optional)

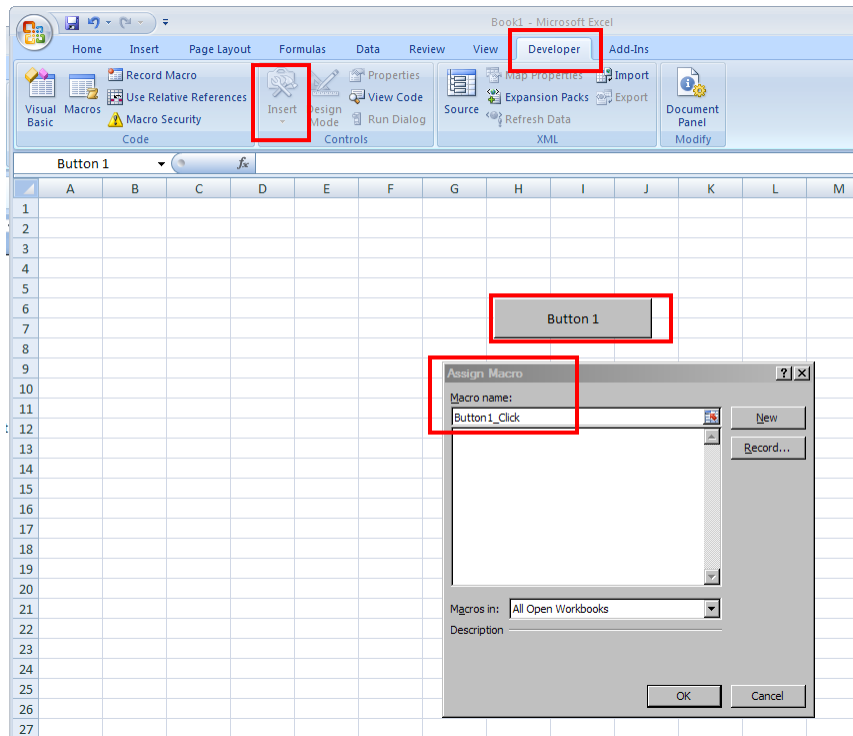
To attach the macro to a GUI button, do the following:

- 1 Click **Developer > Insert**.
- 2 From the **Form Controls** menu, select the **Button (Form Control)** icon.

---

**Tip** Hover your mouse over the Form Controls menu to see the various control labels.

- 3 In the Assign Macros dialog box, select the macro you want to assign the GUI button to, and click **OK**.



### Attaching a Macro to a Button

## For More Information

If you want to...	See...
<ul style="list-style-type: none"><li>• Perform basic MATLAB Programmer tasks</li><li>• Understand how the deployment products process your MATLAB functions</li><li>• Understand how the deployment products work together</li><li>• Explore guidelines about writing deployable MATLAB code</li></ul>	“Write Deployable MATLAB Code” on page 5-10
See more examples about building add-ins and COM components	“Add-in and Macro Creation from MATLAB Functions ” on page 6-5
Learn how to customize and integrate the COM component you built by modifying the Microsoft Visual Basic code	“Code Your Microsoft Visual Basic Application” on page 7-3

## End-to-End Deployment of a MATLAB Function Using the Function Wizard

**In this section...**

- “What Can the Function Wizard Do for Me?” on page 4-3
- “Example File Copying” on page 4-25
- “mymagic Testing” on page 4-25
- “Installation of the Function Wizard” on page 4-4
- “Function Wizard Start-Up” on page 4-27
- “Workflow Selection for Prototyping and Debugging MATLAB Functions” on page 4-28
- “New MATLAB Function Definition” on page 4-30
- “MATLAB Function Prototyping and Debugging” on page 4-41
- “Function Execution from MATLAB” on page 4-43
- “Microsoft Excel Add-in and Macro Creation Using the Function Wizard” on page 4-44
- “Function Execution from the Deployed Component” on page 4-46
- “Macro Execution” on page 4-18
- “Microsoft Excel Add-in and Macro Packaging using the Function Wizard” on page 4-47
- “Microsoft Visual Basic Code Access (Optional Advanced Task)” on page 4-19
- “For More Information” on page 4-49

### MATLAB Programmer

Role	Knowledge Base	Responsibilities
MATLAB programmer	<ul style="list-style-type: none"> <li>• MATLAB expert</li> <li>• No IT experience</li> <li>• No access to IT systems</li> </ul>	<ul style="list-style-type: none"> <li>• Develops models; implements in MATLAB</li> <li>• Uses tools to create a component that is used by the COM developer</li> </ul>

If you are still in the process of developing a MATLAB function that is not yet ready to be deployed, you may find this example to be an appropriate introduction to using MATLAB Builder EX.

The Function Wizard allows you to iteratively test, develop, and debug your MATLAB function. It does this by invoking MATLAB from the Function Wizard Control Panel.

Developing your function in an interactive environment ensures that it works in an expected manner, prior to deployment to larger-scale applications. Usually, these applications are programmed by the Excel Developer using an enterprise language like Microsoft Visual Basic.

Similar to the Magic Square example, the Prototyping and Debugging example develops a function named `mymagic`, which wraps a MATLAB function, `magic`, which computes a magic square, a function with a single multidimensional matrix output.

If your MATLAB function is ready to be deployed and you have already built your add-in and COM component with the Deployment Tool, see “Executing Functions and Creating Macros Using the Function Wizard” on page 4-2.

### Key Tasks for the MATLAB Programmer

Task	Reference
1. Review MATLAB Builder EX prerequisites, if you have not already done so.	“MATLAB Builder EX Prerequisites” on page 1-6
2. Prepare to run the example by copying the example files.	“Example File Copying” on page 4-25
3. Test the MATLAB function you want to deploy as an add-in or COM component.	“mymagic Testing” on page 4-25
4. Install the Function Wizard.	“Installation of the Function Wizard” on page 4-4
5. Start the Function Wizard.	“Function Wizard Start-Up” on page 4-5
6. Select the prototyping and debugging workflow.	“Workflow Selection for Prototyping and Debugging MATLAB Functions” on page 4-28
7. Define the new MATLAB function you want to prototype by adding it to the Function Wizard and establishing input and output ranges.	“New MATLAB Function Definition” on page 4-30
8. Test your MATLAB function by executing it with the Function Wizard.	“Function Execution from MATLAB” on page 4-43

<b>Task</b>	<b>Reference</b>
9. Prototype and Debug the MATLAB function if needed, using MATLAB and the Function Wizard.	“MATLAB Function Prototyping and Debugging” on page 4-41
10. Create the add-in and COM component, as well as the macro, using the Function Wizard to invoke MATLAB and the Deployment Tool.	“Microsoft Excel Add-in and Macro Creation Using the Function Wizard” on page 4-44
11. Execute the your function from the newly created component, to ensure the function's behavior is identical to when it was tested.	“Function Execution from the Deployed Component” on page 4-46
12. Execute the macro you created using the Function Wizard.	“Macro Execution” on page 4-18
13. Package your deployable add-in and macro using the Function Wizard to invoke MATLAB and the Deployment Tool.	“Microsoft Excel Add-in and Macro Packaging using the Function Wizard” on page 4-47
14. Optionally inspect or modify the Microsoft Visual Basic code you generated with the COM component. Optionally, attach the macro you created to a GUI button.	“Microsoft Visual Basic Code Access (Optional Advanced Task)” on page 4-19

## What Can the Function Wizard Do for Me?

The Function Wizard enables you to pass Microsoft Excel (Excel 2000 or later) worksheet values to a compiled MATLAB model and then return model output to a cell or range of cells in the worksheet.

The Function Wizard provides an intuitive interface to Excel worksheets. You do not need previous knowledge of Microsoft Visual Basic for Applications (VBA) programming.

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

---

**Note:** The Function Wizard does not currently support the MATLAB `sparse`, and `complex` data types.

---

## Example File Copying

All MATLAB Builder EX examples reside in `matlabroot\toolbox\matlabxl\examples\`. The following table identifies examples by folder:

For Example Files Relating To...	Find Example Code in Folder...	For Example Documentation See...
Magic Square Example	<code>xlmagic</code>	“Integrate an Add-In and COM Component with Microsoft Excel” on page 1-16
Variable-Length Argument Example	<code>xlmulti</code>	“Work with Variable-Length Inputs and Outputs” on page 6-5
Calling Compiled MATLAB Functions from Microsoft Excel	<code>xlbasic</code>	“Calling Compiled MATLAB Functions from Microsoft Excel” on page 8-4
Spectral Analysis Example	<code>xlspectral</code>	“Build and Integrate with Spectral Analysis” on page 7-16

## mymagic Testing

In this example, you test a MATLAB file (`mymagic.m`) containing the predefined MATLAB function `magic`. You test to have a baseline to compare to the results of the function when it is ready to deploy.

- 1 In MATLAB, locate `mymagic.m`. See “Example File Copying” on page 4-25 for locations of examples. The contents of the file are as follows:

```
function y = mymagic(x)
%MYMAGIC Magic square of size x.
% Y = MYMAGIC(X) returns a magic square of size x.
% This file is used as an example for the MATLAB
% Builder EX product.
```

```
% Copyright 2001-2012 The MathWorks, Inc.
```

```
y = magic(x)
```

- 2 At the MATLAB command prompt, enter `mymagic(5)`. View the resulting output, which appears as follows:



```
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9
```

### Installation of the Function Wizard

Before you can use the Function Wizard, you must first install it as an add-in that is accessible from Microsoft Excel.

After you install the Function Wizard, the entry **MATLAB Functions** appears as an available Microsoft Excel add-in button.

#### Using Office 2007

- 1 Start Microsoft Excel if it is not already running.
- 2  Click the Office Button (  ) and select **Excel Options**.
- 3 In the left pane of the Excel Options dialog box, click **Add-Ins**.
- 4 In the right pane of the Excel Options dialog box, select **Excel Add-ins** from the **Manage** drop-down box.
- 5 Click **Go**.
- 6 Click **Browse**. Navigate to `matlabroot\toolbox\matlabxl\matlabxl\arch` and select `FunctionWizard2007.xlam`. Click **OK**.
- 7 In the Excel Add-Ins dialog box, verify that the entry **MATLAB Builder EX Function Wizard** is selected. Click **OK**.

#### Using Office 2010

- 1 Click the **File** tab.
- 2 On the left navigation pane, select **Options**.
- 3 In the Excel Options dialog box, on the left navigation pane, select **Add-Ins**.



- 4 In the Manage drop-down, select **Excel Add-Ins**, and click **Go**.
- 5 In the Add-Ins dialog box, click **Browse**.
- 6 Browse to *matlabroot/toolbox/matlabxl/matlabxl/arch*, and select *FunctionWizard2007.xlam*. Click **OK**.
- 7 In the Excel Add-Ins dialog, verify that the entry **MATLAB Builder EX Function Wizard** is selected. Click **OK**.

The Home tab of the Microsoft Office Ribbon should now contain the Function Wizard tile. See The Home Tab of the Microsoft Office Ribbon with Function Wizard Installed.

### Using Office 2003

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

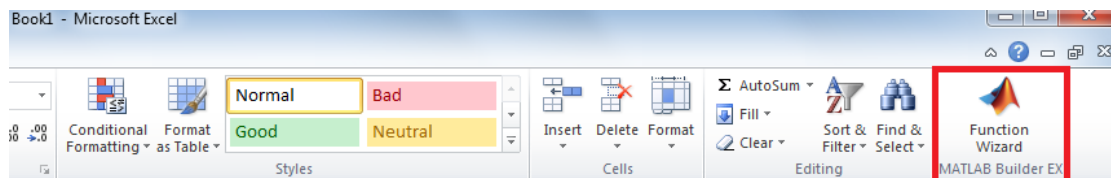
If the Function Wizard was not previously installed, click **Browse** and navigate to *matlabroot\toolbox\matlabxl\matlabxl* folder. Select *FunctionWizard.xla*. Click **OK** to proceed.

## Function Wizard Start-Up

Start the Function Wizard in one of the following ways. When the wizard has initialized, the Function Wizard Start Page dialog box displays.

### Using Office 2007 or Office 2010

In Microsoft Excel, on the Microsoft Office ribbon, on the **Home** tab, select **MATLAB Functions**.

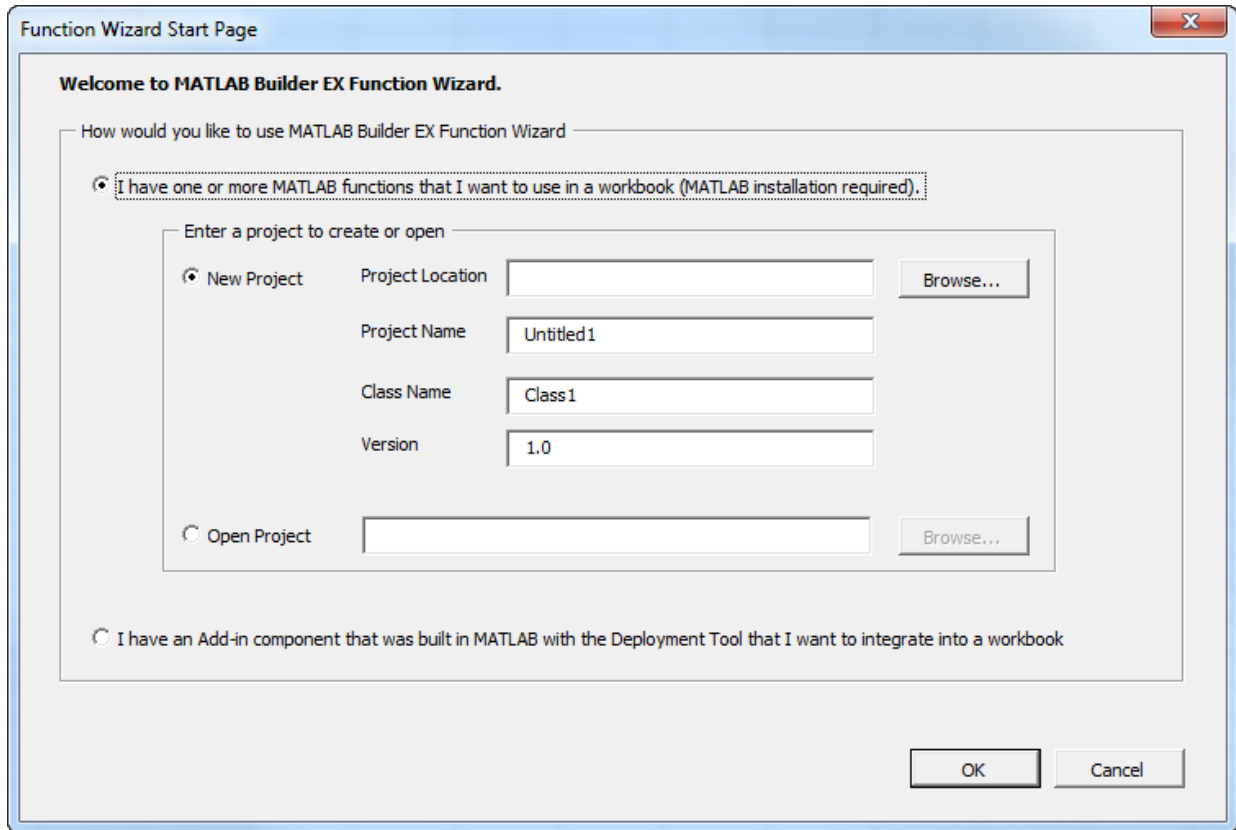


### The Home Tab of the Microsoft Office Ribbon with Function Wizard Installed

#### Using Office 2003

- 1 Select **Tools > Add-Ins** from the Excel main menu.

### 2 Select Function Wizard.



The Function Wizard Start Page Dialog Box

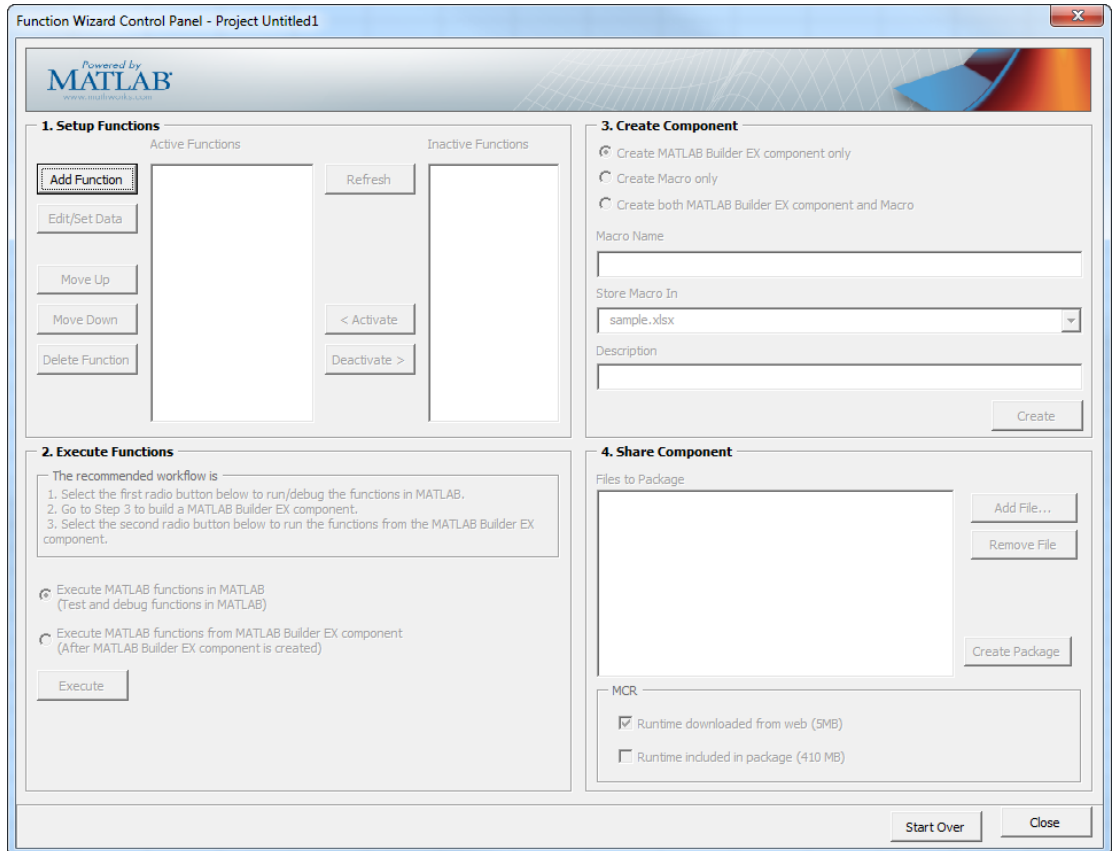
## Workflow Selection for Prototyping and Debugging MATLAB Functions

After you have installed and started the Function Wizard, do the following.

- 1 From the Function Wizard Start Page dialog box, select **I have one or more MATLAB functions that I want to use in a workbook (MATLAB installation required)**. The **New Project** option is selected by default. Enter a meaningful project name in the **Project Name** field, like `testmymagic`, for example.

**Tip** Some customers find it helpful to assign a unique name as the **Class Name** (default is **Class1**) and to assign a **Version** number for source control purposes.

- Click **OK**. The Function Wizard Control Panel displays with the **Add Function** button enabled.



## About Project Files

Keep in mind the following information about project files when working with the Function Wizard:

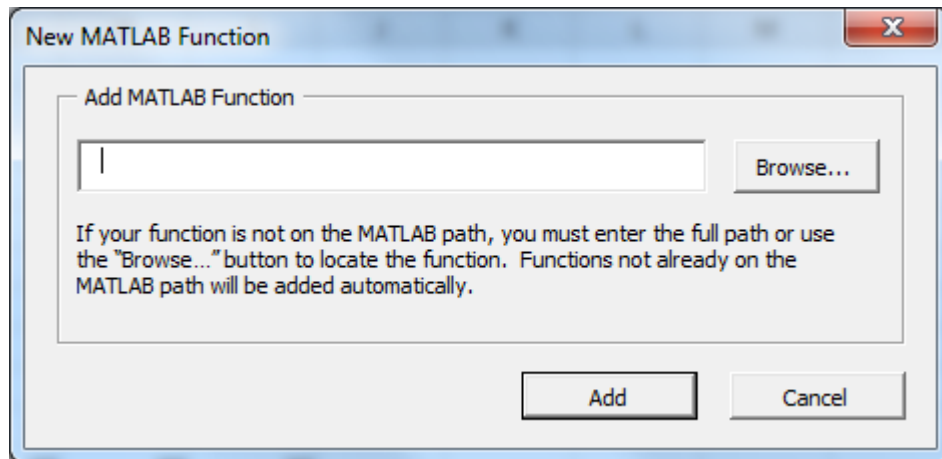
- The project files created by the Function Wizard are the same project files created and used by the Deployment Tool (`deploytool`).
- The Function Wizard prompts you to specify a location for your project files when you open your first new project. Project files are auto-saved to this location and may be opened in the future through either the Deployment Tool or the Function Wizard.
- If you previously built a component using the Function Wizard, the wizard will prompt you to load it.

### Quitting the MATLAB Session Invoked by the Function Wizard

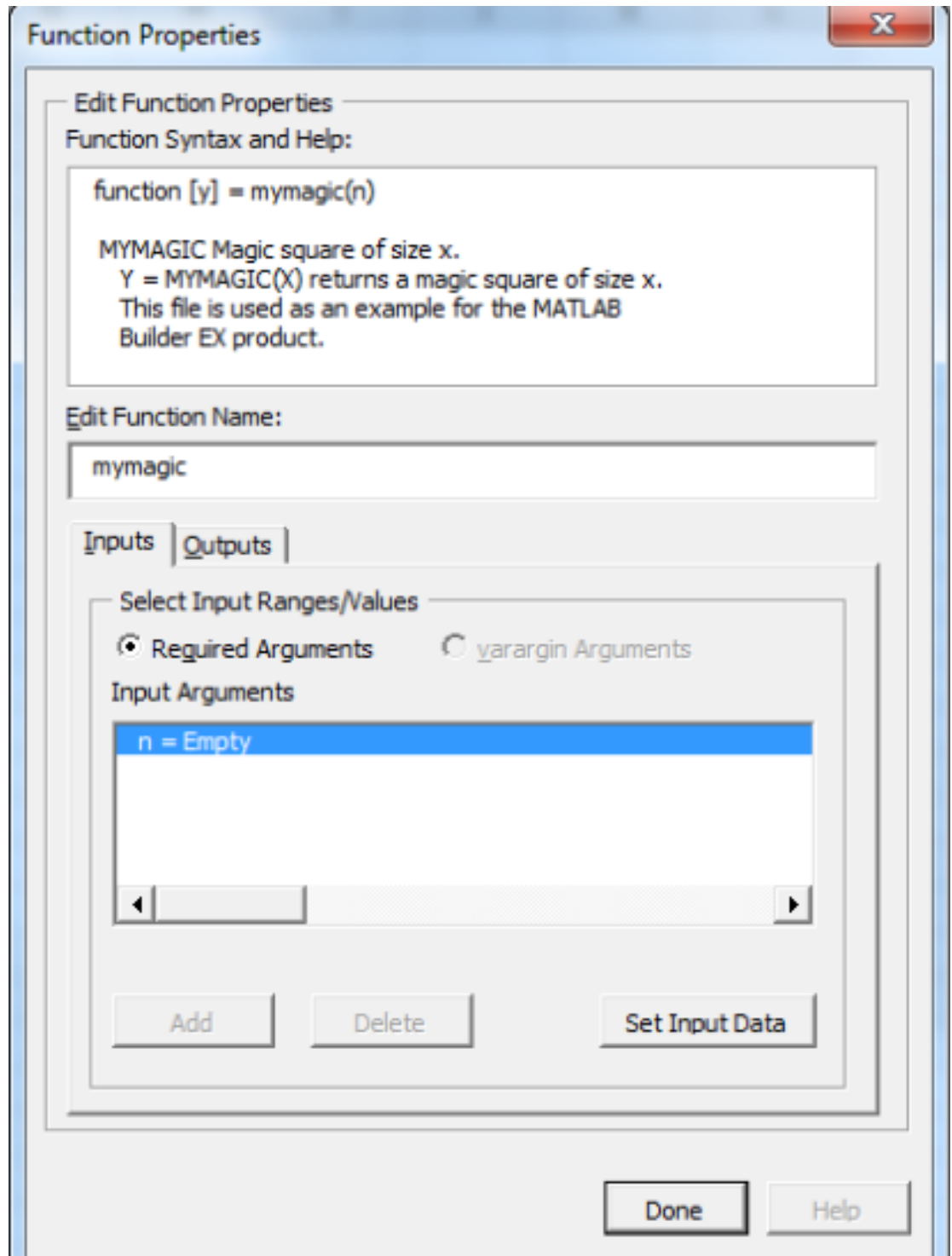
Avoid manually terminating the MATLAB session invoked by the Function Wizard. Doing so can prevent you from using the Wizard's MATLAB-related features from your Excel session. If you want to quit the remotely invoked MATLAB session, restart Excel.

### New MATLAB Function Definition

- 1 Add the function you want to deploy to the Function Wizard. Click **Add** in the Set Up Functions area of the Function Wizard Control Panel. The New MATLAB Function dialog box appears.

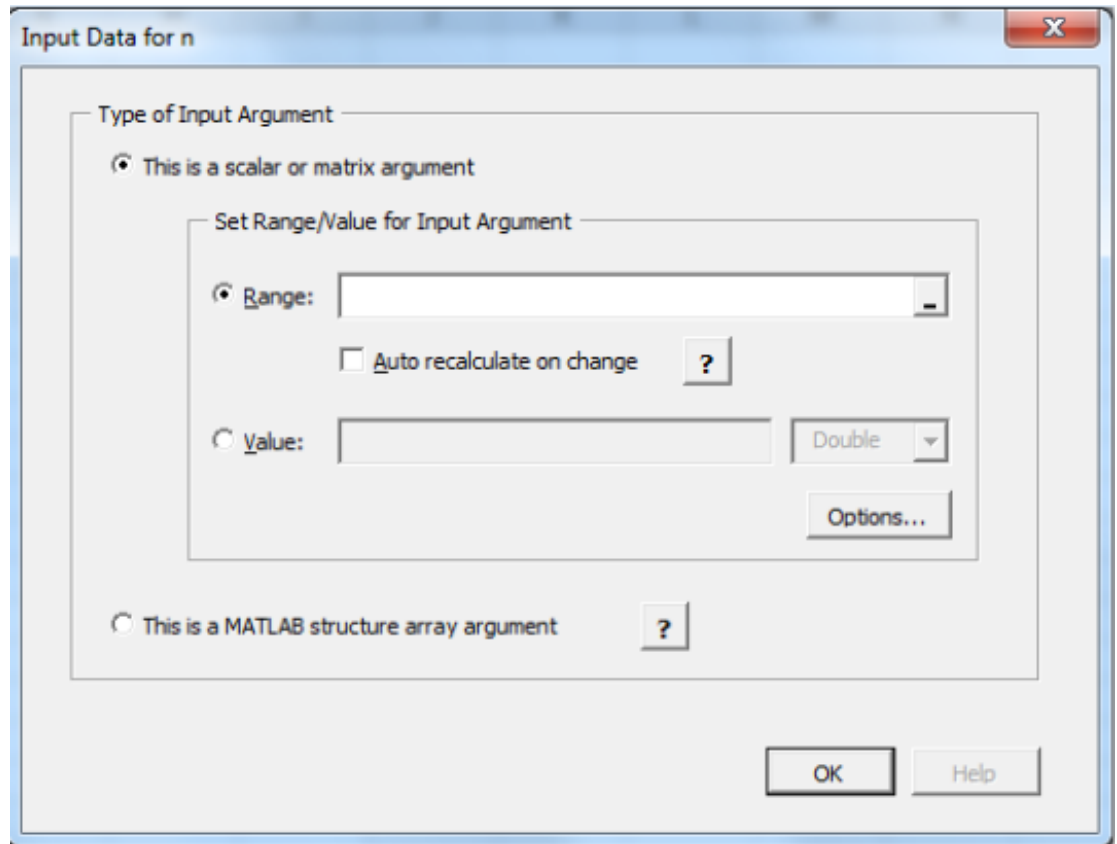


- 2 Browse to locate your MATLAB function. Select the function and click **Open**.
- 3 In the New MATLAB Function dialog box, click **Add**. The Function Properties dialog box appears.



**Tip** The **Function Syntax and Help** area, in the Function Properties dialog box, displays the first help text line (sometimes called the *H1 line*) in a MATLAB function. Displaying these comments in the Function Properties dialog box can be helpful when deploying new or unfamiliar MATLAB functions to end-users.

- 4 Define input argument properties as follows.
  - a On the **Input** tab, click **Set Input Data**. The Input Data for  $n$  dialog box appears.



- b Specify a **Range** or **Value** by selecting the appropriate option and entering the value.

---

**Caution** Avoid selecting ranges using arrow keys. If you must use arrow keys to select ranges, apply the necessary fix from the Microsoft site: <http://support.microsoft.com/kb/291110>.

---

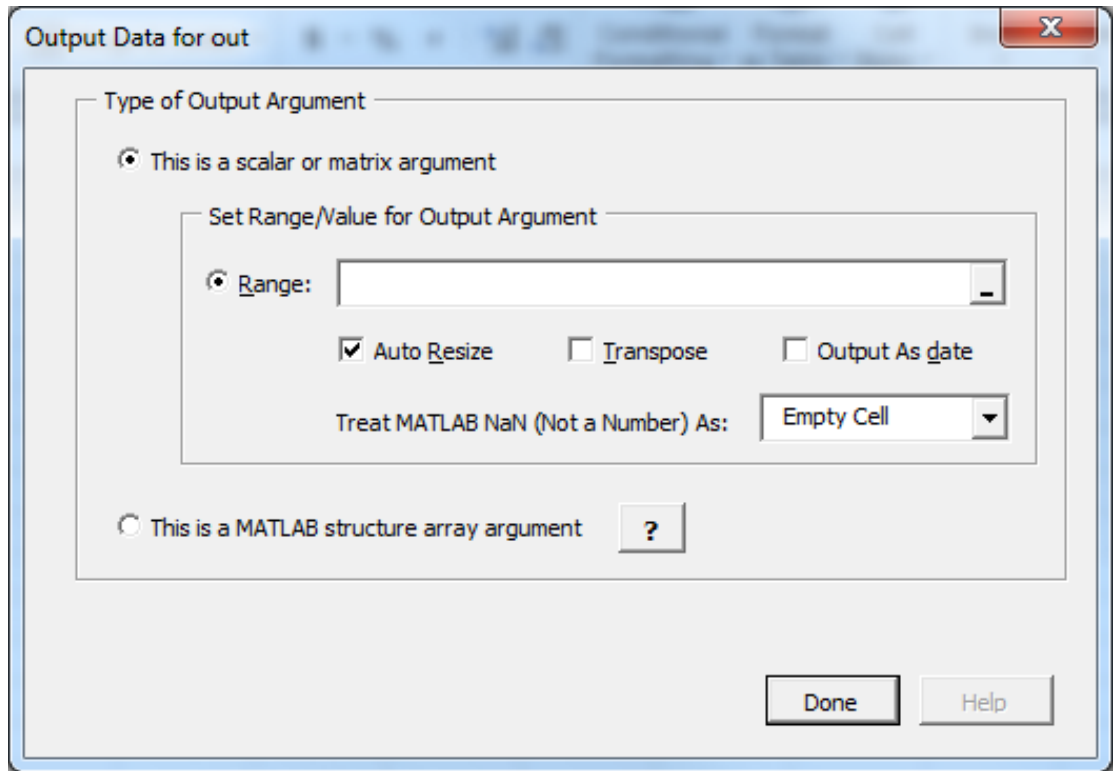
- c Click **Done**.

---

**Tip** To specify how MATLAB Builder EX handles blank cells (or cells containing no data), see “Empty Cell Value Control” on page 4-11.

---

- 5 Define output argument properties as follows.
  - a On the **Output** tab, click **Set Output Data**. The Output Data for  $y$  dialog box appears, where  $x$  is the name of the output variable you are defining properties of.



---

**Tip** You can also specify MATLAB Builder EX to **Auto Resize**, **Transpose** or output your data in date format (**Output as date**). To do so, select the appropriate option in the Argument Properties For *y* dialog box.

---

- b** Specify a **Range**. Alternately, select a range of cells on your Excel sheet; the range will be entered for you in the **Range** field.

---

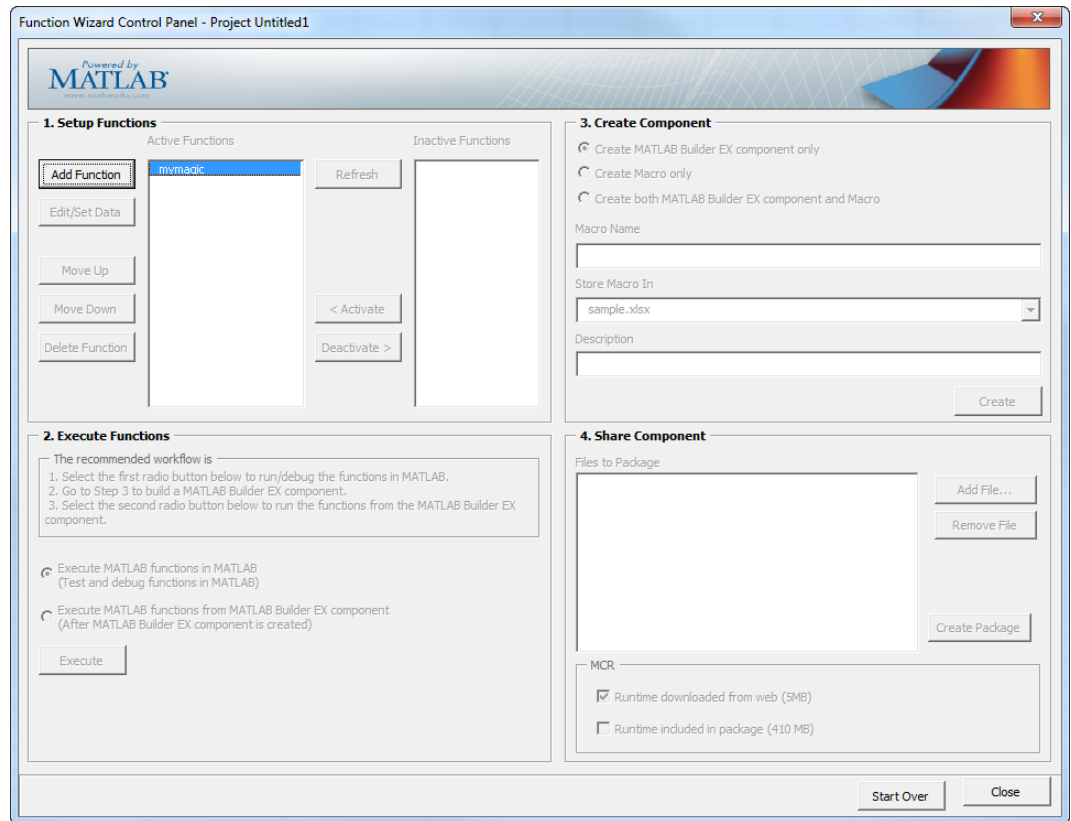
**Caution** Avoid selecting ranges using arrow keys. If you must use arrow keys to select ranges, apply the necessary fix from the Microsoft site: <http://support.microsoft.com/kb/291110>.

---

- c** Select **Auto Resize** if it is not already selected.
- d** Click **Done** in the Argument Properties For *y* dialog box.



- e Click **Done** in the Function Properties dialog box. `mymagic` now appears in the **Active Functions** list of the Function Wizard Control Panel.



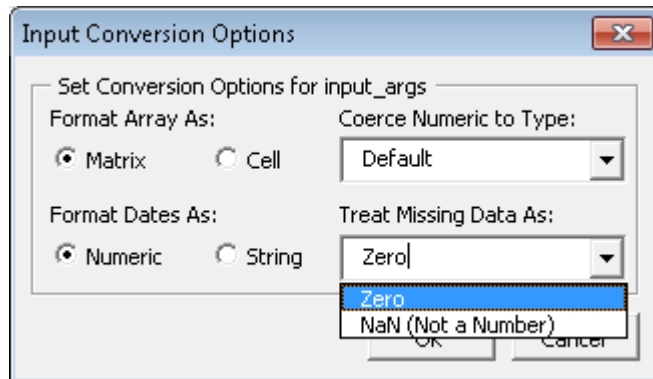
## Empty Cell Value Control

You can specify how MATLAB Builder EX processes empty cells, allowing you to assign undefined or unrepresented (NaN, for example) data values to them.

To specify how to handle empty cells, do the following.

- 1 Click **Options** in the Input Data for  $N$  dialog box.

The Input Conversion Options dialog box opens.

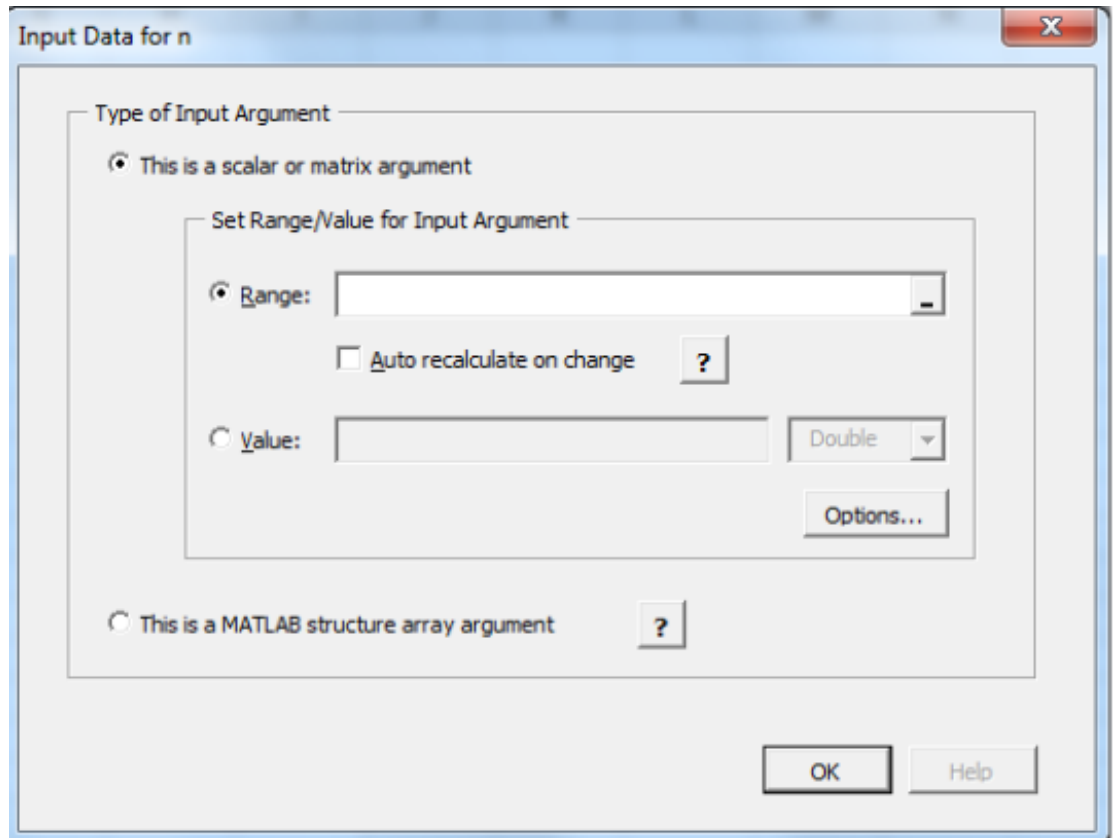


- 2 Click the **Treat Missing Data As** drop-down box.
- 3 Specify either **Zero** or **NaN (Not a Number)**, depending on how you want to handle empty cells.

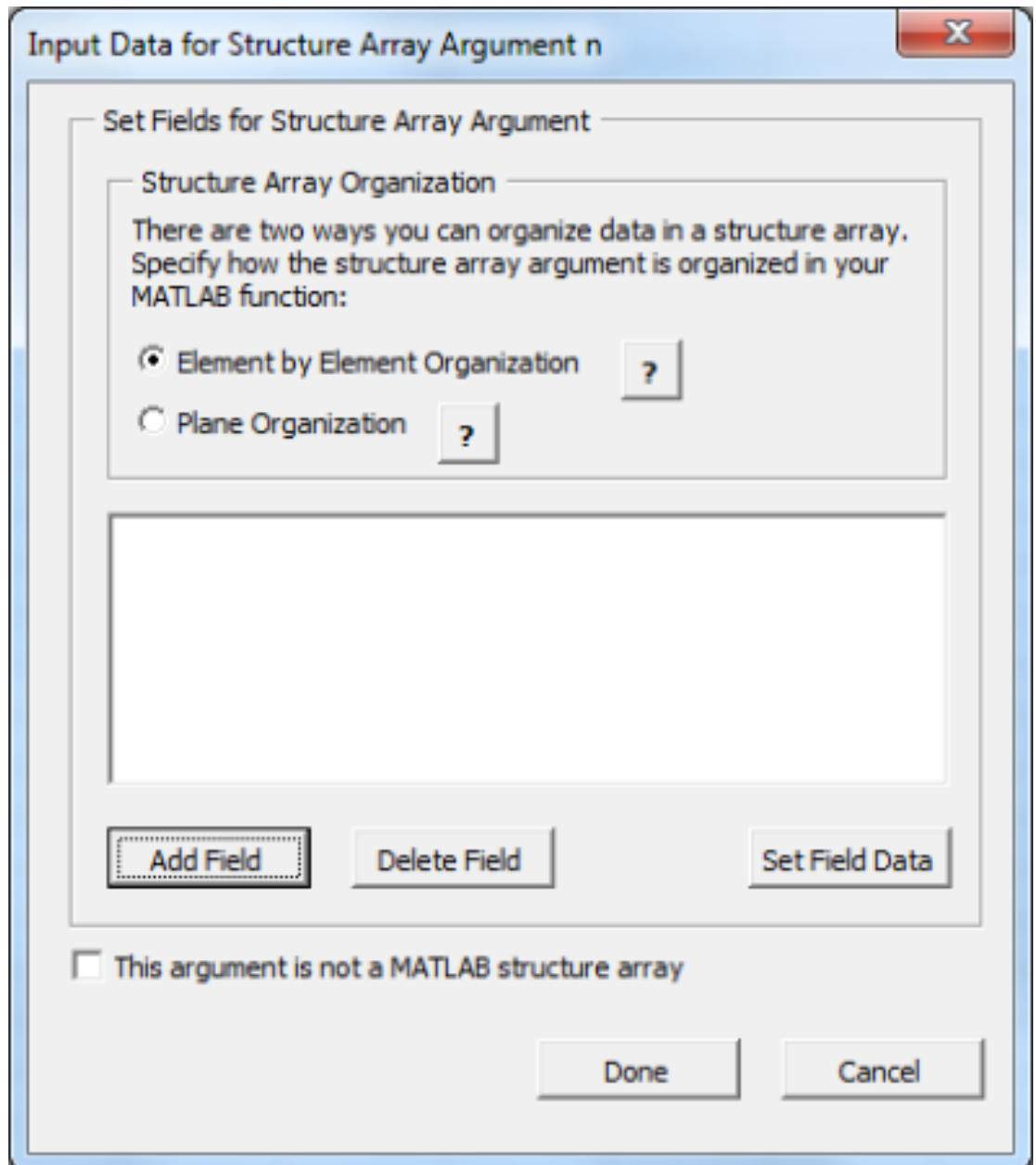
### Working with Struct Arrays

To assign ranges to fields in a struct array, do the following:

- 1 If you have not already done so, select **This is a MATLAB structure array argument** in the Input Data for  $n$  dialog box and click **OK**.



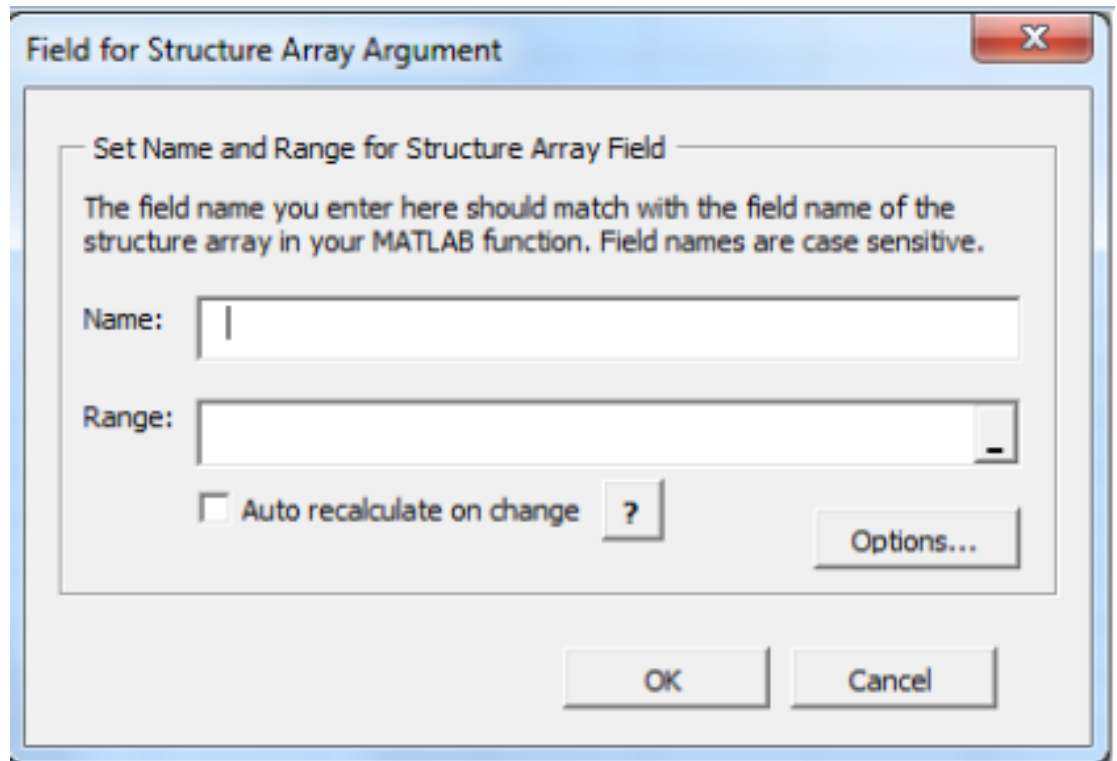
The Input Data for Structure Array Argument  $n$  dialog box opens.



- 2 The Function Wizard supports Vector and Two-dimensional struct arrays organized in either Element by Element or Plane organization, for both input and output.

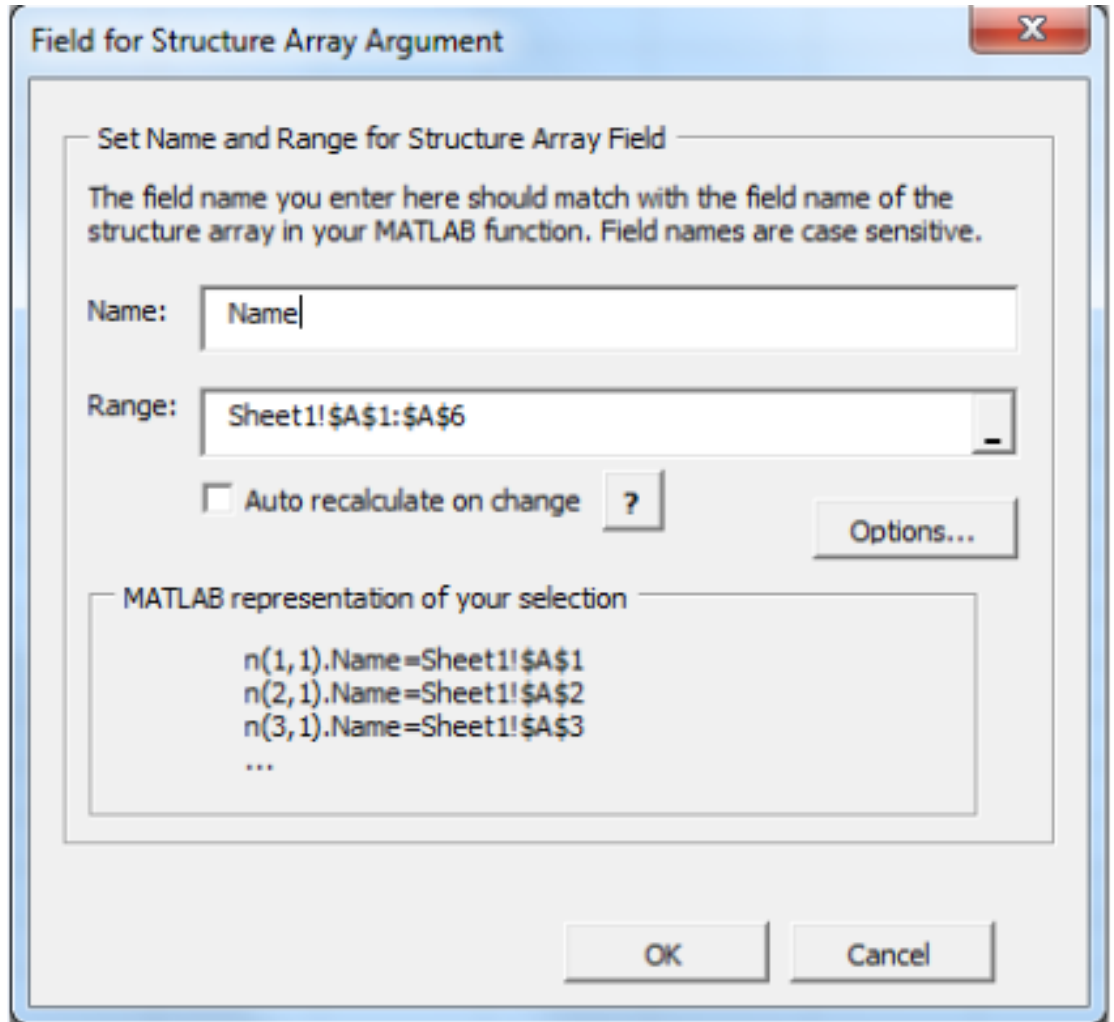
In the Input Data for Structure Array Argument  $n$  dialog box, do the following:

- a In the Structure Array Organization area, select either **Element by Element Organization** or **Plane Organization**.
- b Click **Add Field** to add fields for each of your struct array arguments. The Field for Structure Array Argument dialog box opens.



- 3 In the Field for Argument dialog box, do the following:
  - a In the **Name** field, define the field name. The **Name** you specify must match the field name of the structure array in your MATLAB function.
  - b Specify the **Range** for the field.

**Caution** Avoid selecting ranges using arrow keys. If you must use arrow keys to select ranges, apply the necessary fix from the Microsoft site: <http://support.microsoft.com/kb/291110>.



- c Click **Done**.

### **How Structure Arrays are Supported**

MATLAB Builder EX supports one and two-dimensional MATLAB structure arrays.

The product converts data passed into structure arrays in *element-by-element organization* or *plane organization*. See *MATLAB Programming Fundamentals* for more information about all MATLAB data types, including structures.

### **Deploying Structure Arrays as Inputs and Outputs**

If you are a MATLAB programmer and want to deploy a MATLAB function with structure arrays as input or output arguments, build Microsoft Excel macros using the Function Wizard and pass them (with the Excel Add-In and COM component) to the end users. If you can't do this, let your end users know:

- Which arguments are structure arrays
- Field names of the structure arrays

### **Using Macros with Struct Arrays**

The macro generation feature of MATLAB Builder EX works with struct arrays as input or output arguments. See “Macro Creation” on page 4-18 if you have a MATLAB function you are ready to deploy. See “Microsoft Excel Add-in and Macro Creation Using the Function Wizard” on page 4-44 if you are using the Function Wizard to create your MATLAB function from scratch. See “Choosing Function Deployment Workflow” on page 1-10 for more information on both workflows.

## **MATLAB Function Prototyping and Debugging**

Use the Function Wizard to interactively prototype and debug a MATLAB function.

Since `mymagic` calls the prewritten MATLAB `magic` function directly, it does not provide an illustrative example of how to use the prototyping and debugging feature of MATLAB Builder EX.

Following is an example of how you might use this feature with `myprimes`, a function containing multiple lines of code.

### Prototyping and Debugging with `myprimes`

For example, say you are in the process of prototyping code that uses the equation  $P = \text{myprimes}(n)$ . This equation returns a row vector of prime numbers less than or equal to  $n$  (a prime number has no factors other than 1 and the number itself).

Your code uses  $P = \text{myprimes}(n)$  as follows:

```
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));
```

In designing your code, you want to handle various use cases. For example, you want to experiment with scenarios that may assign a column vector value to the output variable  $p$  (`myprimes` only returns a row vector, as stated previously). You follow this general workflow:

- 1 Set a breakpoint in `myprimes` at the first `if` statement, using the GUI or `dbstop`, for instance.
- 2 On the Function Wizard Control Panel, in the Execute Functions area, click **Execute**. Execution will stop at the previously set breakpoint. Note the value of  $p$  in the yellow box in the following figure. Step-through and debug your code as you normally would using the MATLAB Editor.



The screenshot shows the MATLAB Editor window for the file `myprimes.m`. The code is as follows:

```

1 function [p] = myprimes(n)
2
3 if length(n)~=1, error('N must be a scalar'); end
4 if n < 2, p = zeros(1,0); return, end
5 p = 1:2:n;
6 q = length(p);
7 p(1) = 2;
8 for k = 3:2:sqrt(n)
9
10     if p((k+1)/2)
11         p(((k+1)/2):k:q) = 0;
12     end
13
14 end
15
16 p = (p(p>0));

```

The output of the function is displayed in a yellow box:

```

p: 1x4 double =
     2     3     5     7

```

The status bar at the bottom indicates the current position is Ln 14, Col 4, OVR.

### Debugging myprimes Using the Function Wizard

For more information about debugging MATLAB code, see “Editing and Debugging MATLAB Code” in the *MATLAB Desktop Tools and Development Environment User’s Guide*.

### Function Execution from MATLAB

Test your deployable MATLAB function by executing it in MATLAB:

- 1 From the Function Wizard Control Panel, in the Execute Functions area, select **Execute MATLAB Functions in MATLAB**.

- 2 Click **Execute**. In Excel, the Magic Square function executes, producing results similar to the following.

A	B	C	D	E
17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

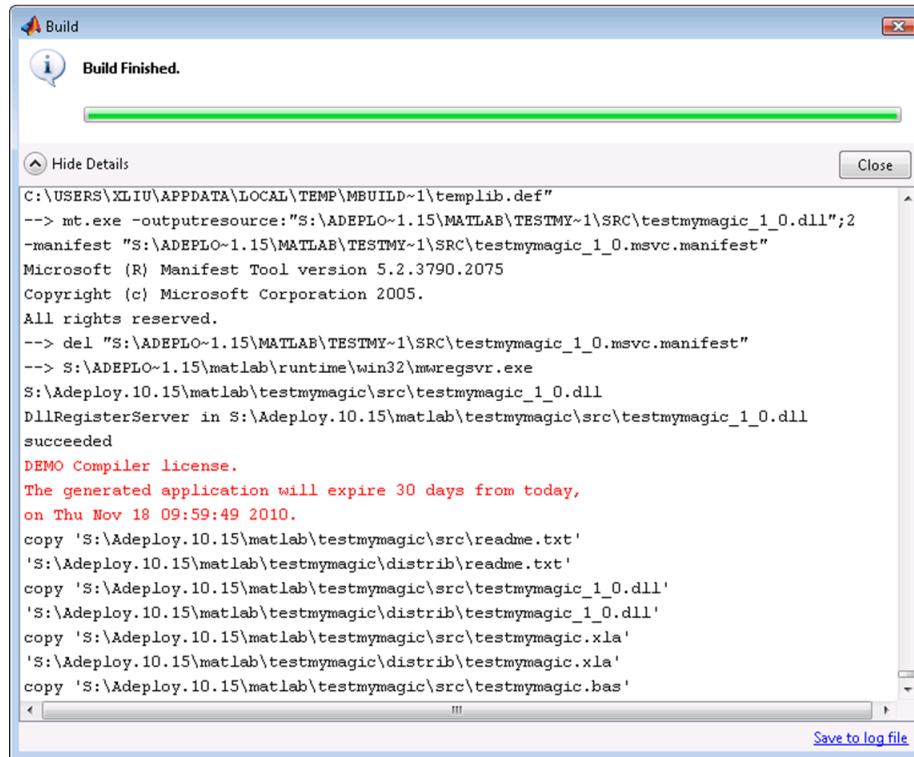
### Microsoft Excel Add-in and Macro Creation Using the Function Wizard

The Function Wizard can automatically create a deployable Microsoft Excel add-in and macro. To create your add-in in this manner, use one of the following procedures.

#### Creating an Add-in and Associated Excel Macro

To create both a deployable add-in and an associated Excel macro:

- 1 In the Function Wizard Control Panel dialog box, in the Create Component area, select **Create Both Builder EX Component and Excel Macro**.
- 2 Enter `mymagic` in the **Macro Name** field.
- 3 Select the location of where to store the macro, using the **Store Macro In** drop-down box.
- 4 Enter a brief description of the macro's functionality in the **Description** field.
- 5 Click **Create** to build both the add-in (as well as the underlying COM component) and the associated macro. The Deployment Tool Build dialog box appears, indicating the status of the add-in and COM component compilation (build).



## The Build Dialog

### Creating a COM Component or a Macro Only Without Creating an Add-in

To create either a COM component or a macro without also creating the Excel add-in, do the following

- 1 In the Function Wizard Control Panel dialog box, in the Create Component area, select either **MATLAB Builder EX Component Only** or **Create Excel Macro Only**.
- 2 Enter **mymagic** in the **Macro Name** field.
- 3 Select the location of where to store the macro, using the **Store Macro In** drop-down box.
- 4 Enter a brief description of the macro's functionality in the **Description** field.

- 5 Click **Create**.

### Function Execution from the Deployed Component

Execute your function as you did similarly in “Function Execution from MATLAB” on page 4-43, but this time execute it from the deployed component to ensure it matches your previous output.

- 1 From the Function Wizard Control Panel, in the Execute Functions area, select **Execute MATLAB Functions from Deployed Component**.
- 2 Click **Execute**. In Excel, the Magic Square function executes, producing results similar to the following.

A	B	C	D	E
17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

### Macro Execution

Run the macro you created in “Macro Creation” on page 4-18 by doing one of the following, after first clearing cells A1:E5 (which contain the output of the Magic Square function you ran in “Function Execution” on page 4-17).

---

**Tip** You may need to enable the proper security settings before running macros in Microsoft Excel. For information about macro permissions and error messages occurring from executing macros, see the Appendix B appendix.

---

#### Using Office 2007 or Office 2010

- 1 In Microsoft Excel, click **View > Macros > View Macros**.
- 2 Select **mymagic** from the **Macro name** drop-down box.

- 3 Click **Run**. Cells A1:E5 on the Excel sheet are automatically populated with the output of `mymagic`.

### Using Office 2003

- 1 In Microsoft Excel, click **Tools > Macro > Macros**.
- 2 Select `mymagic` from the **Macro name** drop-down box.
- 3 Click **Run**. Cells A1:E5 on the Excel sheet are automatically populated with the output of `mymagic`.

## Watch a Video

Watch a video that demonstrates how to execute deployed functions with the Function Wizard and Microsoft Excel.

### Microsoft Excel Add-in and Macro Packaging using the Function Wizard

The Function Wizard can automatically package a deployable Microsoft Excel add-in and macro for sharing. To package your add-in in this manner, use one of the following procedures.

- 1 After successfully building your component and add-in, in the Share Component area of the Function Wizard Control Panel dialog box, review the files listed in the **Files to include in packaging** field. **Add Files** or **Remove Files** to and from the package by clicking the appropriate buttons.
- 2 To add access to the MCR Installer to your package, select one of the options in the MCR area. For information about the MCR and the MCR Installer, see “Install MATLAB Runtime” on page 1-18.
- 3 When you are ready to create your package, click **Create Package**.

### Microsoft Visual Basic Code Access (Optional Advanced Task)

Optionally, you may want to access the Visual Basic code or modify it, depending on your programming expertise or the availability of an Excel developer. If so, follow these steps.

From the Excel main window, open the Microsoft Visual Basic editor by doing one of the following. select **Tools > Macro > Visual Basic Editor**.

### Using Office 2007 or Office 2010

- 1 Click **Developer > Visual Basic**.
- 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 **Matlab Macros** module.

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

### Using Office 2003

- 1 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 **Matlab Macros** module.

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

### Mapping a Macro to a GUI Button or Control (Optional)

To attach the macro to a GUI button, do the following:

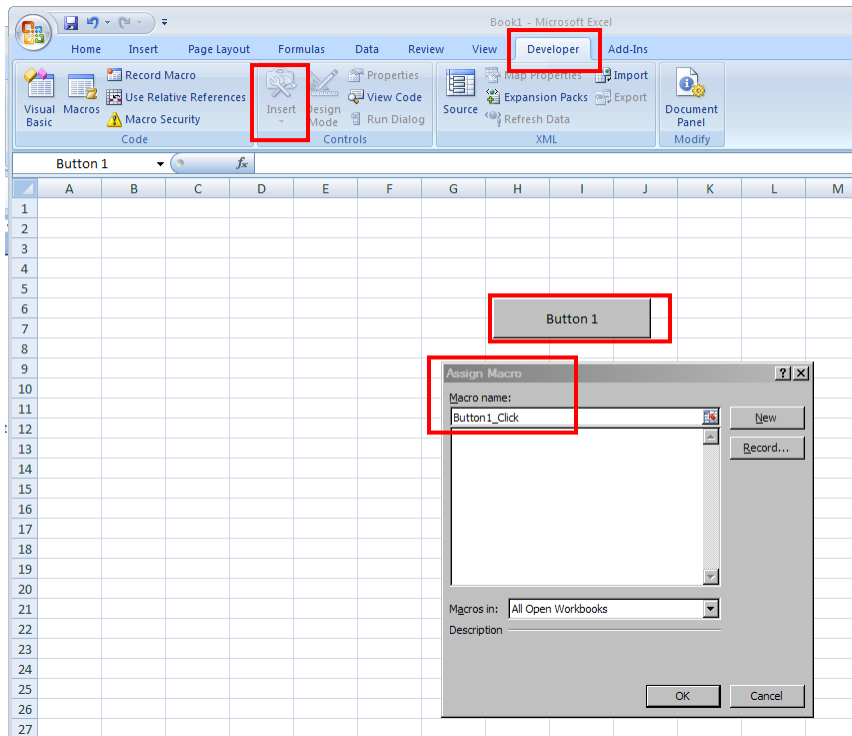
- 1 Click **Developer > Insert**.
- 2 From the **Form Controls** menu, select the **Button (Form Control)** icon.

---

**Tip** Hover your mouse over the **Form Controls** menu to see the various control labels.

---

- 3 In the **Assign Macros** dialog box, select the macro you want to assign the GUI button to, and click **OK**.



### Attaching a Macro to a Button

### For More Information

If you want to...	See...
<ul style="list-style-type: none"> <li>• Perform basic MATLAB Programmer tasks</li> <li>• Understand how the deployment products process your MATLAB functions</li> <li>• Understand how the deployment products work together</li> <li>• Explore guidelines about writing deployable MATLAB code</li> </ul>	<p>“Write Deployable MATLAB Code” on page 5-10</p>

<b>If you want to...</b>	<b>See...</b>
See more examples about building add-ins and COM components	“Add-in and Macro Creation from MATLAB Functions ” on page 6-5
Learn more about the MATLAB runtime	“Working with the MATLAB Runtime”
Learn how to customize and integrate the COM component you built by modifying the Microsoft Visual Basic code	“Code Your Microsoft Visual Basic Application” on page 7-3 “Build and Integrate with Spectral Analysis” on page 7-16



# MATLAB Code Deployment

---

- “Application Deployment Products and the Compiler Apps” on page 5-2
- “Write Deployable MATLAB Code” on page 5-10
- “How the Deployment Products Process MATLAB Function Signatures” on page 5-14
- “Load MATLAB Libraries using loadlibrary” on page 5-16
- “Use MATLAB Data Files (MAT Files) in Compiled Applications” on page 5-18

## Application Deployment Products and the Compiler Apps

### In this section...

“What Is the Difference Between the Compiler Apps and the `mcc` Command Line?” on page 5-2

“How Does MATLAB Compiler Software Build My Application?” on page 5-2

“Dependency Analysis Function” on page 5-5

“MEX-Files, DLLs, or Shared Libraries” on page 5-6

“Deployable Archive” on page 5-6

### What Is the Difference Between the Compiler Apps and the `mcc` Command Line?

When you use one of the compiler apps, you perform any function you would invoke using the MATLAB Compiler `mcc` command-line interface. The compiler apps’ interactive menus and dialogs build `mcc` commands that are customized to your specification. As such, your MATLAB code is processed the same way as if you were compiling it using `mcc`.

Compiler app advantages include:

- You perform related deployment tasks with a single intuitive interface.
- You maintain related information in a convenient project file.
- Your project state persists between sessions.
- You load previously stored compiler projects from a prepopulated menu.
- Package applications for distribution.

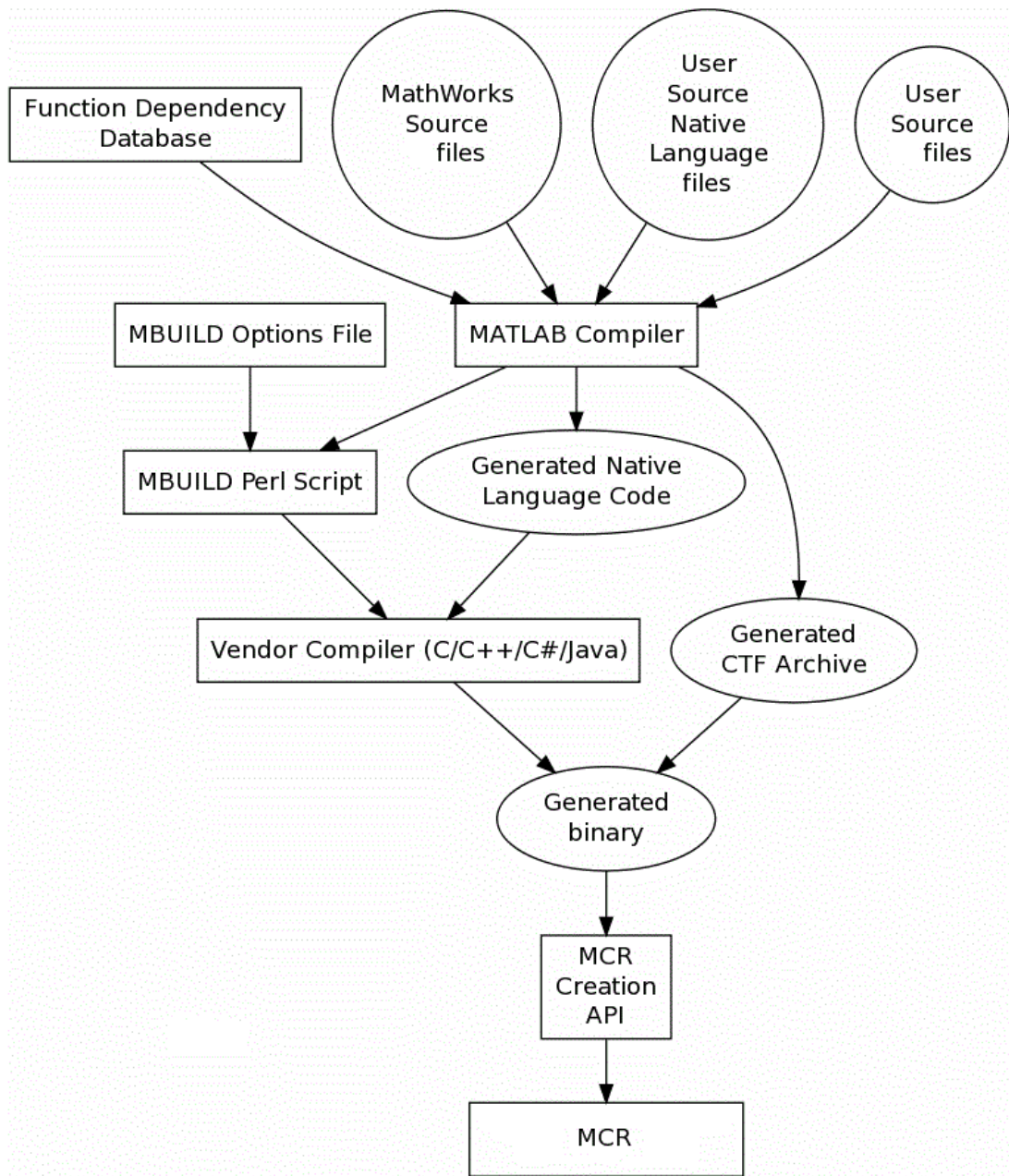
### How Does MATLAB Compiler Software Build My Application?

To build an application, MATLAB Compiler software performs these tasks:

- 1 Parses command-line arguments and classifies by type the files you provide.
- 2 Analyzes files for dependencies using a dependency analysis function. Dependencies affect deployability and originate from functions called by the file. Deployability is affected by:
  - File type — MATLAB, Java, MEX, and so on.

- File location — MATLAB, MATLAB toolbox, user code, and so on.

For more information about how the compiler does dependency analysis, see “Dependency Analysis Function” on page 5-5.



- Circles: Input Files
- Boxes: MathWorks-supplied components
- Ellipses: Output Files

- 3 Validates MEX-files. In particular, `mexFunction` entry points are verified. For more details about MEX-file processing, see “MEX-Files, DLLs, or Shared Libraries” on page 5-6.
- 4 Creates a deployable archive from the input files and their dependencies. For more details about deployable archives see “Deployable Archive” on page 5-6.
- 5 Generates target-specific wrapper code. For example, a C main function requires a very different wrapper than the wrapper for a Java interface class.
- 6 Generates target-specific binary package. For library targets such as C++ shared libraries, Java packages, or .NET assemblies, the compiler will invoke the required third-party compiler.

## Dependency Analysis Function

MATLAB Compiler uses a dependency analysis function to determine the list of necessary files to include in the generated package. Sometimes, this process generates a large list of files, particularly when MATLAB object classes exist in the compilation and the dependency analyzer cannot resolve overloaded methods at compile time. Dependency analysis also processes `include/exclude` files on each pass.

---

**Tip** To improve compile time performance and lessen application size, prune the path with the `mcc` command's `-N` and `-p` flags. You can also specify **Files required for your application** in the compiler app.

---

The dependency analyzer searches for executable content such as:

- MATLAB files
- P-files

---

**Note:** If the MATLAB file corresponding to the p-file is not available, the dependency analysis will not be able to determine the p-file's dependencies.

---

- Java classes and `.jar` files
- `.fig` files
- MEX-files

The dependency analyzer does not search for data files of any kind. You must manually include data files in the search.

## MEX-Files, DLLs, or Shared Libraries

When you compile MATLAB functions containing MEX-files, ensure that the dependency analyzer can find them. Doing so allows you to avoid many common compilation problems. In particular, note that:

- Since the dependency analyzer cannot examine MEX-files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.
- If you have any doubts that the dependency analyzer can find a MATLAB function called by a MEX-file, DLL, or shared library, then manually include that function. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.
- Not all functions are compatible with MATLAB Compiler. Check the file `mccExcludedFiles.log` after your build completes. This file lists all functions called from your application that you cannot deploy.

## Deployable Archive

Each application or shared library you produce using MATLAB Compiler has an embedded deployable archive. The archive contains all the MATLAB based content (MATLAB files, MEX-files, and so on). All MATLAB files in the deployable archive are encrypted using the Advanced Encryption Standard (AES) cryptosystem.

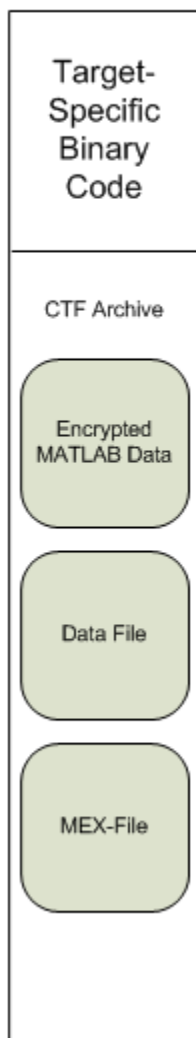
If you choose to extract the deployable archive as a separate file, the files remain encrypted. For more information on how to extract the deployable archive refer to the references in the following table.

### Information on Deployable Archive Embedding/Extraction and Component Cache

Product	Refer to
MATLAB Compiler	“MATLAB Runtime Component Cache and Deployable Archive Embedding”
MATLAB Builder NE	“MATLAB Runtime Component Cache and Deployable Archive Embedding”
MATLAB Builder JA	“Deployable Archive Embedding and Extraction”

<b>Product</b>	<b>Refer to</b>
MATLAB Builder EX	“Using MCR Component Cache and CTF Archive Embedding”

## Generated Component (EXE, DLL, SO, etc)





### **Additional Details**

Multiple deployable archives, such as those generated with COM components, .NET assemblies, or Excel add-ins, can coexist in the same user application. You cannot, however, mix and match the MATLAB files they contain. You cannot combine encrypted and compressed MATLAB files from multiple deployable archives into another deployable archive and distribute them.

All the MATLAB files from a given deployable archive associate with a unique cryptographic key. MATLAB files with different keys, placed in the same deployable archive, do not execute. If you want to generate another application with a different mix of MATLAB files, recompile these MATLAB files into a new deployable archive.

MATLAB Compiler deletes the deployable archive and generated binary following a failed compilation, but only if these files did not exist before compilation initiates. Run `help mcc -K` for more information.

---

**Caution Release Engineers and Software Configuration Managers:** Do not use build procedures or processes that strip shared libraries on deployable archives. If you do, you can possibly strip the deployable archive from the binary, resulting in run-time errors for the driver application.

---

## Write Deployable MATLAB Code

### In this section...

“Compiled Applications Do Not Process MATLAB Files at Runtime” on page 5-10

“Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files” on page 5-11

“Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths” on page 5-11

“Gradually Refactor Applications That Depend on Noncompilable Functions” on page 5-12

“Do Not Create or Use Nonconstant Static State Variables” on page 5-12

“Get Proper Licenses for Toolbox Functionality You Want to Deploy” on page 5-13

### Compiled Applications Do Not Process MATLAB Files at Runtime

MATLAB Compiler secures your code against unauthorized changes. Deployable MATLAB files are suspended or frozen at the time MATLAB Compiler encrypts them—they do not change from that point onward. This does not mean that you cannot deploy a flexible application—it means that *you must design your application with flexibility in mind*. If you want the end user to be able to choose between two different methods, for example, both methods must be available in the deployable archive.

The MATLAB runtime only works on MATLAB code that was encrypted when the deployable archive was built. Any function or process that dynamically generates new MATLAB code will not work against the MATLAB runtime.

Some MATLAB toolboxes, such as the Neural Network Toolbox™ product, generate MATLAB code dynamically. Because the MATLAB runtime only executes encrypted MATLAB files, and the Neural Network Toolbox generates unencrypted MATLAB files, some functions in the Neural Network Toolbox cannot be deployed.

Similarly, functions that need to examine the contents of a MATLAB function file cannot be deployed. HELP, for example, is dynamic and not available in deployed mode. You can use LOADLIBRARY in deployed mode if you provide it with a MATLAB function prototype.

Instead of compiling the function that generates the MATLAB code and attempting to deploy it, perform the following tasks:

- 1 Run the code once in MATLAB to obtain your generated function.
- 2 Compile the MATLAB code with MATLAB Compiler, including the generated function.

---

**Tip** Another alternative to using `EVAL` or `FEVAL` is using anonymous function handles.

---

If you require the ability to create MATLAB code for dynamic run time processing, your end users must have an installed copy of MATLAB.

## Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files

In general, good programming practices advise against redirecting a program search path dynamically within the code. Many developers are prone to this behavior since it mimics the actions they usually perform on the command line. However, this can lead to problems when deploying code.

For example, in a deployed application, the MATLAB and Java paths are fixed and cannot change. Therefore, any attempts to change these paths (using the `cd` command or the `addpath` command) fails

If you find you cannot avoid placing `addpath` calls in your MATLAB code, use `ismcc` and `isdeployed`. See “Use `ismcc` and `isdeployed` Functions To Execute Deployment-Specific Code Paths” on page 5-11 for details.

## Use `ismcc` and `isdeployed` Functions To Execute Deployment-Specific Code Paths

The `isdeployed` function allows you to specify which portion of your MATLAB code is deployable, and which is not. Such specification minimizes your compilation errors and helps create more efficient, maintainable code.

For example, you find it unavoidable to use `addpath` when writing your `startup.m`. Using `ismcc` and `isdeployed`, you specify when and what is compiled and executed.

## Gradually Refactor Applications That Depend on Noncompilable Functions

Over time, refactor, streamline, and modularize MATLAB code containing non-compilable or non-deployable functions that use `ismcc` and `isdeployed`. Your eventual goal is “graceful degradation” of non-deployable code. In other words, the code must present the end user with as few obstacles to deployment as possible until it is practically eliminated.

Partition your code into design-time and run time code sections:

- *Design-time code* is code that is currently evolving. Almost all code goes through a phase of perpetual rewriting, debugging, and optimization. In some toolboxes, such as the Neural Network Toolbox product, the code goes through a period of self-training as it reacts to various data permutations and patterns. Such code is almost never designed to be deployed.
- *Run-time code*, on the other hand, has solidified or become stable—it is in a finished state and is ready to be deployed by the end user.

Consider creating a separate directory for code that is not meant to be deployed or for code that calls undeployable code.

## Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

- Global variables in MATLAB code
- Static variables in MEX-files
- Static variables in Java code

The state of these variables is persistent and shared with everything in the process.

When deploying applications, using persistent variables can cause problems because the MATLAB runtime process runs in a single thread. You cannot load more than one of these non-constant, static variables into the same process. In addition, these static variables do not work well in multithreaded applications.

When programming against compiled MATLAB code, you should be aware that an instance of the MATLAB runtime is created for each instance of a new class. If the same class is instantiated again using a different variable name, it is attached to the MATLAB

runtime created by the previous instance of the same class. In short, if an assembly contains  $n$  unique classes, there will be maximum of  $n$  instances of MATLAB runtimes created, each corresponding to one or more instances of one of the classes.

If you must use static variables, bind them to instances. For example, defining instance variables in a Java class is preferable to defining the variable as `static`.

---

**Note:** This guideline does not apply to MATLAB Builder EX. When programming with Microsoft Excel, you can assign global variables to large matrices that persist between calls.

---

## Get Proper Licenses for Toolbox Functionality You Want to Deploy

You must have a valid MathWorks® license for toolboxes you use to create deployable MATLAB code.

## How the Deployment Products Process MATLAB Function Signatures

In this section...
“MATLAB Function Signature” on page 5-14
“MATLAB Programming Basics” on page 5-14

### MATLAB Function Signature

MATLAB supports multiple signatures for function calls.

The generic MATLAB function has the following structure:

```
function [Out1,Out2,...,varargout]=foo(In1,In2,...,varargin)
```

To the *left* of the equal sign, the function specifies a set of explicit and optional return arguments.

To the *right* of the equal sign, the function lists explicit *input* arguments followed by one or more optional arguments.

All arguments represent a specific MATLAB type.

When the compiler or builder product processes your MATLAB code, it creates several overloaded methods that implement the MATLAB functions. Each of these overloaded methods corresponds to a call to the generic MATLAB function with a specific number of input arguments.

In addition to these methods, the builder creates another method that defines the return values of the MATLAB function as an input argument. This method simulates the `feval` external API interface in MATLAB.

### MATLAB Programming Basics

## Creating a Deployable MATLAB Function

Virtually any calculation that you can create in MATLAB can be deployed, if it resides in a function. For example:

```
>> 1 + 1
```

cannot be deployed.

However, the following calculation:

```
function result = addSomeNumbers()  
    result = 1+1;  
end
```

can be deployed because the calculation now resides in a function.

## Taking Inputs into a Function

You typically pass inputs to a function. You can use primitive data type as an input into a function.

To pass inputs, put them in parentheses. For example:

```
function result = addSomeNumbers(number1, number2)  
    result = number1 + number2;  
end
```

## Load MATLAB Libraries using loadlibrary

---

**Note:** It is important to understand the difference between the following:

- MATLAB `loadlibrary` function — Loads shared library into MATLAB.
  - Operating system `loadlibrary` function — Loads specified Windows or UNIX operating system module into the address space of the calling process.
- 

With MATLAB Compiler version 4.0 (R14) and later, you can use MATLAB file prototypes as described below to load your library in a compiled application. Loading libraries using H-file headers is not supported in compiled applications. This behavior occurs when `loadlibrary` is compiled with the header argument as in the statement:

```
loadlibrary(library, header)
```

In order to work around this issue, execute the following at the MATLAB command prompt:

```
loadlibrary(library, header, 'mfilename', 'mylibrarymfile');
```

where *mylibrarymfile* is the name of a MATLAB file you would like to use when loading this library. This step only needs to be performed once to generate a MATLAB file for the library.

In the code that is to be compiled, you can now call `loadlibrary` with the following syntax:

```
loadlibrary(library, @mylibrarymfile, 'alias', alias)
```

It is only required to add the prototype `.m` file and `.dll` file to the deployable archive of the deployed application. There is no need for `.h` files and C/C++ compilers to be installed on the deployment machine if the prototype file is used.

Once the prototype file is generated, add the file to the deployable archive of the application being compiled. You can do this with the `-a` option (if using the `mcc` command) or by dragging it under **Other/Additional Files** (as a helper file) if using one of the compiler apps.

With MATLAB Compiler versions 4.0.1 (R14+) and later, generated MATLAB files will automatically be included in the deployable archive as part of the compilation process.



For MATLAB Compiler versions 4.0 (R14) and later, include your library MATLAB file in the compilation with the `-a` option with `mcc`.

## **Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler**

Note the following limitations in regards to using `loadlibrary` with MATLAB Compiler. For complete documentation and up to date restrictions on `loadlibrary`, please reference the MATLAB documentation.

- You can not use `loadlibrary` inside of MATLAB to load a “shared library built with MATLAB Compiler”.
- With MATLAB Compiler Version 3.0 (R13SP1) and earlier, you cannot compile calls to `loadlibrary` because of general restrictions and limitations of the product.

## Use MATLAB Data Files (MAT Files) in Compiled Applications

### In this section...

“Explicitly Including MAT files Using the `%#function` Pragma” on page 5-18

“Load and Save Functions” on page 5-18

“MATLAB Objects” on page 5-21

### Explicitly Including MAT files Using the `%#function` Pragma

MATLAB Compiler excludes MAT files from “Dependency Analysis Function” on page 5-5 by default.

If you want MATLAB Compiler to explicitly inspect data within a MAT file, you need to specify the `%#function` pragma when writing your MATLAB code.

For example, if you are creating a solution with Neural Network Toolbox, you need to use the `%#function` pragma within your GUI code to include a dependency on the `gmdistribution` class, for instance.

### Load and Save Functions

If your deployed application uses MATLAB data files (MAT-files), it is helpful to code `LOAD` and `SAVE` functions to manipulate the data and store it for later processing.

- Use `isdeployed` to determine if your code is running in or out of the MATLAB workspace.
- Specify the data file by either using `WHICH` (to locate its full path name) define it relative to the location of `ctfroot`.
- All MAT-files are unchanged after `mcc` runs. These files are not encrypted when written to the deployable archive.

For more information about deployable archives, see “Deployable Archive” on page 5-6.

See the `ctfroot` reference page for more information about `ctfroot`.

Use the following example as a template for manipulating your MATLAB data inside, and outside, of MATLAB.

## Using Load/Save Functions to Process MATLAB Data for Deployed Applications

The following example specifies three MATLAB data files:

- user\_data.mat
- userdata\extra\_data.mat
- ..\externdata\extern\_data.mat

**1** Navigate to *matlab\_root*\extern\examples\compiler\Data\_Handling.

**2** Compile ex\_loadsave.m with the following mcc command:

```
mcc -mv ex_loadsave.m -a 'user_data.mat' -a
    '\userdata\extra_data.mat' -a
    '..\externdata\extern_data.mat'
```

### ex\_loadsave.m

```
function ex_loadsave
% This example shows how to work with the
% "load/save" functions on data files in
% deployed mode. There are three source data files
% in this example.
%   user_data.mat
%   userdata\extra_data.mat
%   ..\externdata\extern_data.mat
%
% Compile this example with the mcc command:
%   mcc -m ex_loadsave.m -a 'user_data.mat' -a
%   '\userdata\extra_data.mat'
%   -a '..\externdata\extern_data.mat'
% All the folders under the current main MATLAB file directory will
%   be included as
% relative path to ctroot; All other folders will have the
%   folder
% structure included in the deployable archive file from root of the
%   disk drive.
%
% If a data file is outside of the main MATLAB file path,
%   the absolute path will be
% included in deployable archive and extracted under ctroot. For example:
%   Data file
%   "c:\$matlabroot\examples\externdata\extern_data.mat"
%   will be added into deployable archive and extracted to
%   "$ctroot\$matlabroot\examples\externdata\extern_data.mat".
%
% All mat/data files are unchanged after mcc runs. There is
% no encryption on these user included data files. They are
% included in the deployable archive.
%
% The target data file is:
%   .\output\saved_data.mat
```

```
% When writing the file to local disk, do not save any files
% under ctfroot since it may be refreshed and deleted
% when the application isnext started.

%==== load data file =====
if isdeployed
    % In deployed mode, all file under CTFroot in the path are loaded
    % by full path name or relative to $ctfroot.
    % LOADFILENAME1=which(fullfile(ctfroot,mfilename,'user_data.mat'));
    % LOADFILENAME2=which(fullfile(ctfroot,'userdata','extra_data.mat'));
    LOADFILENAME1=which(fullfile('user_data.mat'));
    LOADFILENAME2=which(fullfile('extra_data.mat'));
    % For external data file, full path will be added into deployable archive;
    % you don't need specify the full path to find the file.
    LOADFILENAME3=which(fullfile('extern_data.mat'));
else
    %running the code in MATLAB
    LOADFILENAME1=fullfile(matlabroot,'extern','examples','compiler',
        'Data_Handling','user_data.mat');
    LOADFILENAME2=fullfile(matlabroot,'extern','examples','compiler',
        'Data_Handling','userdata','extra_data.mat');
    LOADFILENAME3=fullfile(matlabroot,'extern','examples','compiler',
        'externdata','extern_data.mat');
end

% Load the data file from current working directory
disp(['Load A from : ',LOADFILENAME1]);
load(LOADFILENAME1,'data1');
disp('A= ');
disp(data1);

% Load the data file from sub directory
disp(['Load B from : ',LOADFILENAME2]);
load(LOADFILENAME2,'data2');
disp('B= ');
disp(data2);

% Load extern data outside of current working directory
disp(['Load extern data from : ',LOADFILENAME3]);
load(LOADFILENAME3);
disp('ext_data= ');
disp(ext_data);

%==== multiple the data matrix by 2 =====
result = data1*data2;
disp('A * B = ');
disp(result);

%==== save the new data to a new file =====
SAVEPATH=strcat(pwd,filesep,'output');
if ( ~isdir(SAVEPATH))
    mkdir(SAVEPATH);
end
SAVEFILENAME=strcat(SAVEPATH,filesep,'saved_data.mat');
disp(['Save the A * B result to : ',SAVEFILENAME]);
```

```
save(SAVEFILENAME, 'result');
```

## **MATLAB Objects**

When working with MATLAB objects, remember to include the following statement in your MAT file:

```
%#function class_constructor
```

Using the `%#function` pragma in this manner forces the dependency analysis to load needed class definitions, enabling the “MATLAB runtime” to successfully load the object.



# Microsoft Excel Add-in Creation, Function Execution, and Deployment

---

- “Supported Compilation Targets” on page 6-2
- “The Library Compiler and the Command Line Interface” on page 6-4
- “Add-in and Macro Creation from MATLAB Functions ” on page 6-5
- “Execute Add-in With the Function Wizard” on page 6-11

## Supported Compilation Targets

In this section...
“Microsoft Excel Add-in” on page 6-2
“What Are MATLAB Builder EX Components and When Should You Create Them?” on page 6-2
“MATLAB Builder EX Limitations” on page 6-3

### Microsoft Excel Add-in

Using MATLAB Builder EX and the Deployment Tool (`deploytool` or `mcc`), you create deployable add-ins and Excel functions from MATLAB code that runs in Microsoft Excel applications. The Deployment Tool and `mcc` support the following function outputs:

- No outputs
- Figure (graphical) output
- Scalar output

Using the MATLAB Builder EX Function Wizard, you create Excel macros that can be deployed as add-ins or workbooks.

MATLAB Builder EX supports multidimensional matrix output(s).

### Migration Considerations for 32-bit and 64-bit Microsoft Excel

Add-ins created with MATLAB Builder are not compatible with versions of Microsoft Excel that do not share the same bitness:

- Add-ins created with 64-bit versions MATLAB Builder EX are *only* compatible with 64-bit versions of Microsoft Excel.
- Add-ins created with 32-bit versions MATLAB Builder EX are *only* compatible with 32-bit versions of Microsoft Excel.

## What Are MATLAB Builder EX Components and When Should You Create Them?

MATLAB Builder EX generates two primary artifacts from your MATLAB code, using the Deployment Tool: a Microsoft Excel add-in and a COM component.



Existing MATLAB functions can be integrated into new or existing spreadsheet solutions, as add-ins, with the “Function Wizard”. You can also use the Function Wizard to “create and test add-ins” for new MATLAB functions.

COM components created with either the Deployment Tool or the Function Wizard (.bas files) can be integrated into existing Microsoft Visual Basic applications.

MATLAB Builder EX Components are ideal solutions for deploying MATLAB code to users of Microsoft Excel and large-scale enterprise financial applications written in Microsoft Visual Basic.

### **MATLAB Builder EX Limitations**

MATLAB objects are not supported as inputs or outputs for compiled or deployed functions with MATLAB Builder EX.

## The Library Compiler and the Command Line Interface

<b>In this section...</b>
“Using the Library Compiler” on page 6-4
“Using the mcc Command Line Interface” on page 6-4

### Using the Library Compiler

For a complete example of how to use the Library Compiler app, see “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12. For information about how the graphical interface differs from the command line interface, read “What Is the Difference Between the Compiler Apps and the mcc Command Line?” in “Write Deployable MATLAB Code” on page 5-10.

### Using the mcc Command Line Interface

As an alternative to using the Library Compiler, you can use the command line to create your deployable executable, using the `mcc` command.

# Add-in and Macro Creation from MATLAB Functions

**In this section...**

“Create Add-ins and Macros with Single and Multiple Outputs” on page 6-5

“Work with Variable-Length Inputs and Outputs” on page 6-5

## Create Add-ins and Macros with Single and Multiple Outputs

The basic workflow for Microsoft Excel add-in and macro creation can be found in “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12 and “Integrate an Add-In and COM Component with Microsoft Excel”.

## Work with Variable-Length Inputs and Outputs

This example shows you how to work with, and create macros from, functions having variable-length inputs and outputs

### Overview

`myplot` takes a single integer input and plots a line from 1 to that number.

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

`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.

### Where Is the Example Code?

See “Example File Copying” on page 4-25 for information about accessing the example code from within the product.

### Creating the Project

Use the following information as you work through this example using the instructions in “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12:

Project Name	xlmulti
Class Name	xlmulticlass
File to compile (in the xlmulti folder of myfiles\work)	myplot.m myprimes.m mysum.m

**Adding a MATLAB Builder EX COM Function to Microsoft Excel**

- 1 Start Microsoft Excel on your system.
- 2 Open the file `myfiles\work\xlmulti\xlmulti.xls`.

The example appears as shown:

	A	B	C	D	E	F	G	H	I	J	K
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											
9											
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											
32											
33											
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										

**Note** If an Excel prompt says that this file contains macros, click **Enable Macros** to run this example. See the "Troubleshooting" appendix in this User's Guide for details on enabling macros.

## 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 MATLAB. 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.

## 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	11	
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**.

### 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 selecting **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 (Optional)

- 1 On the Microsoft Excel main window, select **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.

### For More Information

For more information about working with variable-length arguments, see “Programming with Variable Arguments” on page 7-8.



## Execute Add-in With the Function Wizard

### In this section...

“Execute an Add-in to Validate Nongraphical Function Output” on page 6-11

“Execute a Graphical Function” on page 6-11

“Create Dialog Box and Error Message Macros” on page 6-15

### Execute an Add-in to Validate Nongraphical Function Output

If you have built your add-in and COM component using `deploytool` or `mcc` and are ready to begin validating your non-graphical function's output, see “Executing Functions and Creating Macros Using the Function Wizard” on page 4-2.

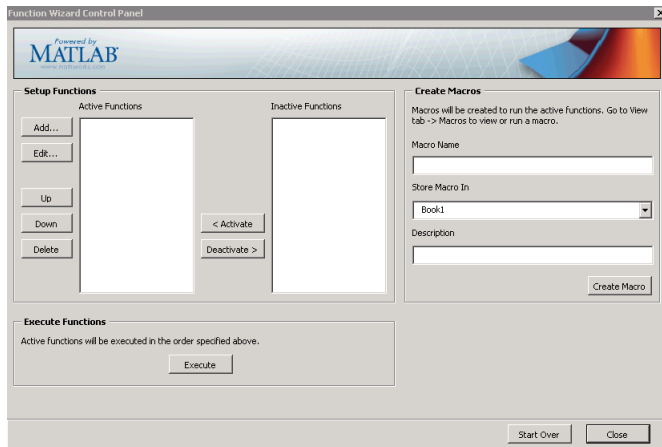
#### Functions Having Multiple Outputs

When working with functions having multiple outputs, simply define each specific output range with the Argument Properties For  $y$  dialog box. See the Argument Properties For  $y$  step in “Function Execution” on page 4-17 for details.

### Execute a Graphical Function

Execute a graphical function on a Microsoft Excel spreadsheet by doing the following.

- 1 Install and start the Function Wizard using the procedures detailed in “Installation of the Function Wizard” on page 4-4 and “Function Wizard Start-Up” on page 4-5. Successfully completing each of these procedures causes the Function Wizard Control Panel to display.

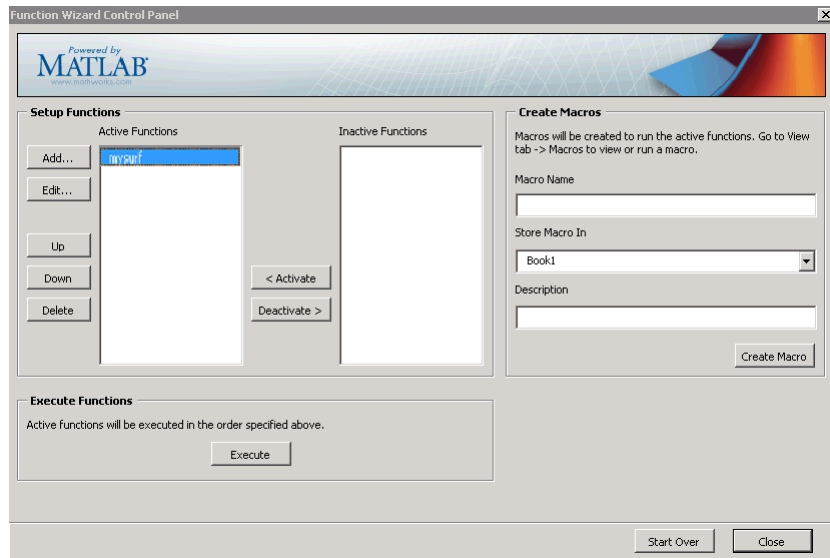


- 2 Click **Add**.
- 3 Select a function with a graphical output, such as **mysurf** for example, from the **Functions for Class *class\_name*** box.
- 4 Click **Add**. The Function Properties dialog box appears.
- 5 Click **Done**. The Function Wizard Control Panel appears with **mysurf** selected in the list of **Active Functions**.

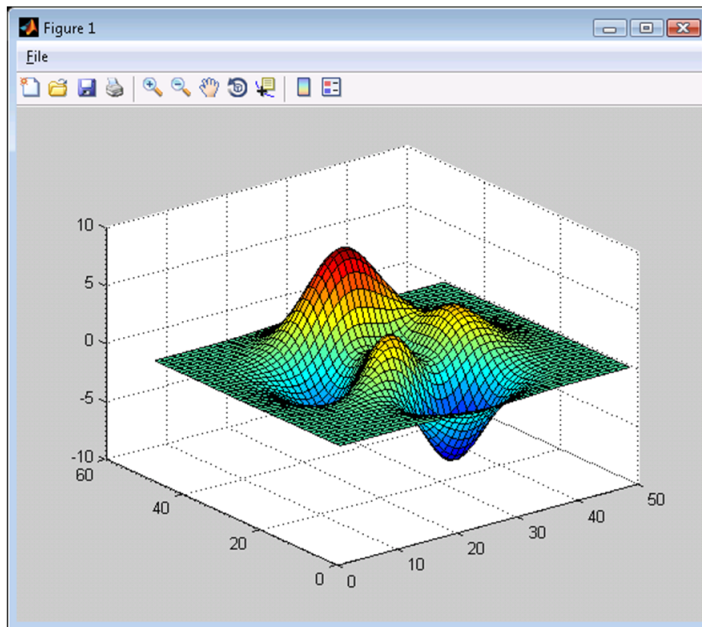
---

**Note:** Since **mysurf.m** does not have any inputs or outputs, there is no need to specify **Properties**.

---



- 6 In the Execute Functions area of the Function Wizard Control Panel, click **Execute**. The graphical output for `mysurf` appears in a separate window.



- 7 Test to ensure you can interact with the figure and that it is usable.

For example, try dragging the figure window, inserting color bars and legends in the toolbar, and so on.

If you encounter problems working with the figure, consult the person who created it (usually the MATLAB Programmer).

### Create a Macro Using a Graphical Function

Once you are satisfied your graphical figure is usable, do the following to create a macro to execute it at your convenience.

---

**Caution** To create a macro, you must have already built your COM component and add-in with MATLAB Builder EX.

For complete Function Wizard workflows, see “Executing Functions and Creating Macros Using the Function Wizard” on page 4-2 and “End-to-End Deployment of a MATLAB Function Using the Function Wizard” on page 4-22.

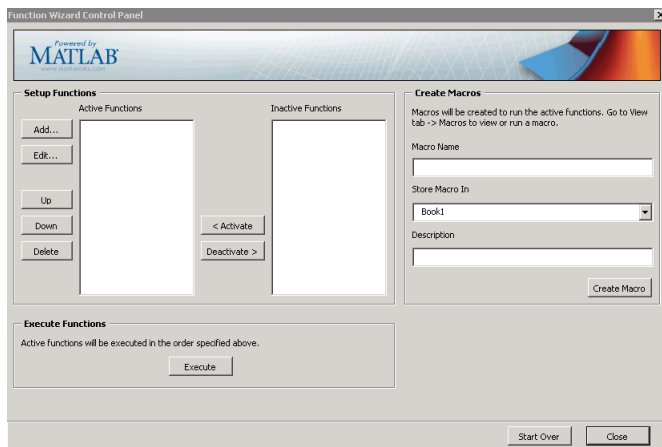
---

- 1 In the Function Wizard Control Panel, label the macro by entering `mysurf` in the **Macro Name** field of the Create Macros area.
- 2 If desired, change the default value **Book1** (for the default Excel sheet name) in the **Store Macro In** field.
- 3 Click **Create Macro**.
- 4 See “Macro Execution” on page 4-18 for details on executing macros with different versions of Microsoft Office. When the macro is **Run**, you should see output similar to the Surf Peaks image in “Execute a Graphical Function” on page 6-11 in this chapter.

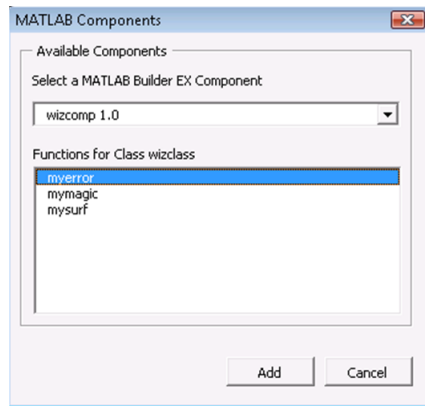
## Create Dialog Box and Error Message Macros

Create a macro that displays a dialog box using this workflow, which is useful for error message presentation.

- 1 Install and start the Function Wizard using the procedures detailed in “Installation of the Function Wizard” on page 4-4 and “Function Wizard Start-Up” on page 4-5. Successfully completing each of these procedures causes the Function Wizard Control Panel to display.



- 2 Click **Add**. The MATLAB Components dialog box appears.



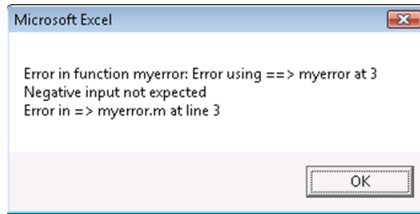
- 3 Select a function that displays a graphical error message, such as **myerror** for example, from the **Functions for Class *class\_name*** box.
- 4 Click **Add**. The Function Properties dialog box appears.
- 5 Associate an input value of -1 with **myerror**.
  - a On the **Inputs** tab, click **Properties**. The Argument Properties for in dialog box appears.
  - b Select **Value** and enter -1.
  - c Click **Done**.
- 6 Define the output of **myerror**—any Excel spreadsheet cell, in this case.
  - a On the **Outputs** tab, click **Properties**. The Argument Properties For *x* dialog box appears, where *x* is the name of the output variable you are defining properties of.
  - b Select **Range** and enter and spreadsheet cell value, =C13, for example.
  - c Click **Done**. The Function Wizard Control Panel appears with **myerror** selected in the list of **Active Functions**.

---

**Tip** If you have functions besides **myerror** listed in the **Active Functions** list that you don't want to execute when you test **myerror**, deactivate these functions by selecting them and clicking **Deactivate**.

---

- 7 Click **Execute**. The following will display.



### Create a Macro That Displays an Error Message or Dialog Box

Create a macro to display your error message on demand.

- 1 In the Function Wizard Control Panel, label the macro by entering `myerror` in the **Macro Name** field of the Create Macros area.
- 2 If desired, change the default value **Book1** (for the default Excel sheet name) in the **Store Macro In** field.
- 3 Click **Create Macro**.
- 4 See “Macro Execution” on page 4-18 for details on executing macros with different versions of Microsoft Office.





# Microsoft Excel Add-in Integration

---

- “Overview of the Integration Process ” on page 7-2
- “Code Your Microsoft Visual Basic Application” on page 7-3
- “Build and Integrate with Spectral Analysis” on page 7-16
- “For More Information” on page 7-30

## 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 you have the skill set of a and have a working knowledge of VBA.

This section is not intended to teach you to program in Visual Basic. Refer to the VBA documentation provided with Excel for general programming information.

For a comprehensive example of building and integrating a COM component using Visual Basic programming, see “Build and Integrate with Spectral Analysis” on page 7-16.

# Code Your Microsoft Visual Basic Application

**In this section...**

“When to Use a Formula Function or a Subroutine” on page 7-3

“Initializing MATLAB Builder EX Libraries with Microsoft Excel” on page 7-3

“Creating an Instance of a Class” on page 7-4

“Calling the Methods of a Class Instance” on page 7-7

“Programming with Variable Arguments” on page 7-8

“Modifying Flags” on page 7-10

“Handling Errors During a Method Call” on page 7-15

## 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 zero outputs.

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, MATLAB Builder EX 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 MATLAB Builder EX Libraries with Microsoft Excel

Before you use any MATLAB Builder EX component, initialize the supporting libraries with the current instance of Microsoft 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 “Class `MWUtil`” on page 10-3.

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 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

- “Overview” on page 7-5
- “CreateObject Function” on page 7-5
- “New Operator” on page 7-5
- “How the MATLAB Runtime Is Shared Among Classes” on page 7-6

## 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 programming interface (API) `CreateObject` function to create an instance of the class. Microsoft refers to calling `CreateObject` as *late binding* and using `new` as *early binding*.

To use this method, declare a variable of type `Object` using `Dim` 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 can be dimensioned as simply `myclass`. The full declaration in the form `<component-name>. <class-name>` guards against name collisions that can occur if other libraries in the current project contain types named `myclass`.

Using both `CreateObject` and `New` produce a dimensioned class instance. The first method does not require a reference to the type library in the VBA project; 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 Microsoft Visual Basic 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 MATLAB Runtime Is Shared Among Classes

MATLAB Builder EX creates a single MATLAB runtime when the first Microsoft COM class is instantiated in an application. This MATLAB runtime is reused and shared

among all subsequent class instances within the component, resulting in more efficient memory usage and eliminating the MATLAB runtime startup cost in each subsequent class instantiation.

All class instances share a single MATLAB workspace and share global variables in the MATLAB 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. MATLAB Builder EX applies a standard mapping from the original MATLAB function syntax to the method's argument list. See “Referencing Utility Classes” on page 10-2 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
    Set aClass = New mycomponent.myclass
    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
```

## Programming with Variable Arguments

### 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 As Variant
    Dim y As Variant
    Dim MCLUtil As Object

    On Error GoTo Handle_Error
    set aClass = CreateObject("mycomponent.myclass.1_0")
    Set MCLUtil = CreateObject("MWComUtil.MWUtil")
    Call MCLUtil.MWPack(v, x1, x2, x3, x4, x5)
    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 “Class MWUtil” on page 10-3 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("MWComUtil.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

```

### Passing an Empty varargin from Microsoft Visual Basic Code

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

### Passing an Empty varargin from VBA Code

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

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

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

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

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

### For More Information

For more information about working with variable-length arguments, see “Work with Variable-Length Inputs and Outputs” on page 6-5.

### Modifying Flags

- “Overview” on page 7-11

- “Array Formatting Flags” on page 7-11
- “Data Conversion Flags” on page 7-13

## 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`” 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 7-5.

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 MATLAB Builder EX , 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 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, MATLAB realizes one-dimensional arrays as 1-by-n matrices (row vectors) that become rows in an Excel worksheet.

---

**Tip** If your MATLAB function is specifically returning a row vector, for example, ensure you assign a similar row vector of cells in Excel.

---

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 MATLAB. 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

## 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.

## Build and Integrate with Spectral Analysis

In this section...
“Overview” on page 7-16
“Building the Component” on page 7-16
“Integrating the Component Using VBA” on page 7-18
“Testing the Add-in” on page 7-25
“Packaging and Distributing the Add-in” on page 7-27
“Installing the Add-in” on page 7-29

### 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` — 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
- `plotfft` — performs the same operations as `computefft`, but also plots the input data and the power spectral density in a MATLAB Figure window

---

**Note:** The MATLAB code for these two functions resides in two MATLAB files, `computefft.m` and `plotfft.m`.

---

The code for `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)));
```

The code for `plotfft.m.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')
```

Build the COM component using the Library Compiler app and the following settings. To proceed with the actual building of the component:

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

See the instructions in “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12 for more information.

### Where Is the Example Code?

See “Example File Copying” on page 4-25 for information about accessing the example code from within the product.

## Integrating the Component Using VBA

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

---

**Note:** To use `Fourier.xla` directly in the folder `xlspectral` (see “Example File Copying” on page 4-25) add references to **Fourier 1.0 Type Library** and **MWComUtil 7.X Type Library**.

---

### Selecting the Libraries

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, select **Tools > Macro > Visual Basic Editor**.
- 3 When the Visual Basic Editor starts, select **Tools > References** to open the Project References dialog box.
- 4 Select **Fourier 1.0 Type Library** and **MWComUtil 7.x 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:

- 1 Right-click the **VBAProject** item in the project window and select **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**.
- 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.Fourier '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. To create a new user form and populate it with the necessary controls:

- 1 Right-click **VBAProject** in the VBA project window and select **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.
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.

Control Type	Control Name	Properties	Purpose
Label	Label1	Caption = Input Data:	Labels the RefEdit for input data.
TextBox	edtSample	Not applicable	Not applicable
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.
Label	Label6	Caption = Power Spectral Density	Labels the RefEdit for power spectral density.
RefEdit	refedtInput	Not applicable	Selects range for input data.
RefEdit	refedtFreq	Not applicable	Selects output range for frequency points.
RefEdit	refedtReal	Not applicable	Selects output range for real part of FFT of input data.
RefEdit	refedtImag	Not applicable	Selects output range for imaginary part of FFT of input data.
RefEdit	refedtPowSpect	Not applicable	Selects output range for power spectral density of input data.

#### 4 When the form and controls are complete, right-click the form and select **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. 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
        refedtInput.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 theFFTData 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")
    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, 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.

To implement the menu item:

- 1 Right-click the **ThisWorkbook** item in the VBA project window and select **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 CommandBarButton  
    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 into the <project - folder>\for\_testing folder that Library Compiler created when building the project.

<project - folder> refers to the project folder that Library Compiler 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, select **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-folder>\for_testing`.
- 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, select **Tools > Add-Ins**.
- 3** When the Add-Ins dialog box appears, click **Browse**.
- 4** Browse to the `<project-folder>\for_testing` folder, 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

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

$= \text{SIN}(2*\text{PI}()*15*A1) + \text{SIN}(2*\text{PI}()*40*A1) + \text{RAND}()$

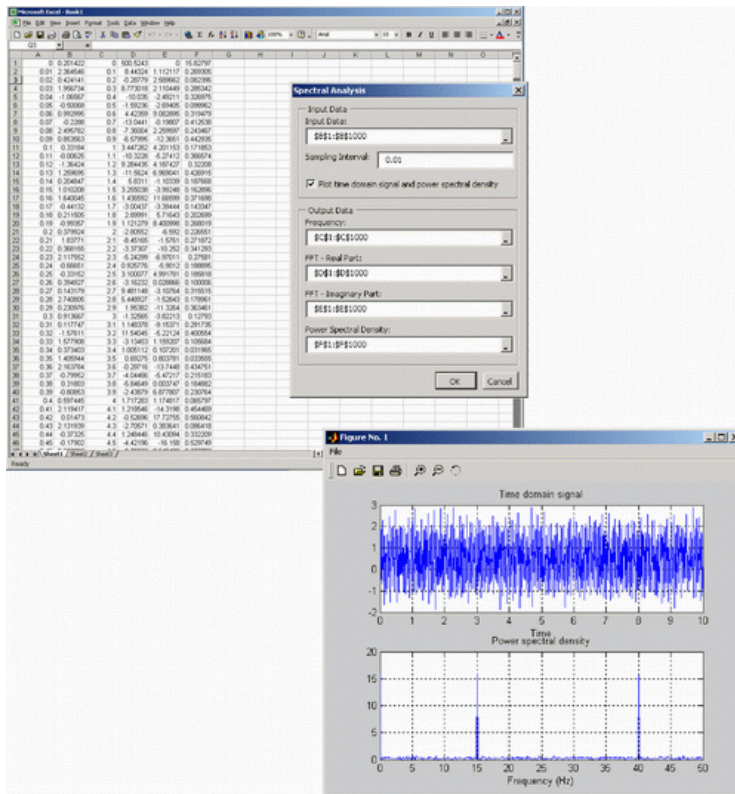
Repeat the drag procedure to copy this formula to all cells in the range B1:B1001.

### 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 the Spectral Analysis add-in.

- 1 On the **Main File** section of the tool strip choose one of the two options described in the following table.

Option	What Does This Option Do?	When Should I Use This Option?
<b>Runtime downloaded from web</b>	The MCR installer downloads the MCR from the Mathwork's Web site.	<ul style="list-style-type: none"> <li>• You have a large number of end users who deploy applications frequently</li> <li>• Your users have internet/network access</li> <li>• Resources such as disk space, performance, and processing time are significant concerns for your organization</li> </ul>
<b>Runtime included in package</b>	The MCR is included in the generated installer. The MCR installer uses the included MCR.	<ul style="list-style-type: none"> <li>• You have a limited number of end users who deploy a small number of applications at sporadic intervals</li> <li>• Your users have no internet/network access</li> <li>• Resources such as disk space, performance, and processing time are not significant concerns</li> </ul> <p><b>Note:</b> Distributing the MCR with the application requires more resources.</p>

For more information about the role the MCR plays in the deployment process, see “Distributing MATLAB Code Using the MATLAB Runtime” on page 1-18.

- 2 Add others files you feel may be useful to end users.

To package additional files or folders, add them to the **Files installed with your application** field. See “Specify Files to Install with the Application” on page 2-8.

- 3 Click Package.

## Installing the Add-in

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.

## For More Information

<b>For more information about...</b>	<b>See...</b>
Functions available through MATLAB Builder EX	Function reference in MATLAB Builder EX documentation
Utility functions you can use to customize COM components created with MATLAB Builder EX	“Referencing Utility Classes” on page 10-2

# Distribution to End Users

---

- “Distribute Your Add-Ins and COM Components to End Users” on page 8-2
- “Deploying Your Microsoft Visual Basic Application” on page 8-4
- “For More Information” on page 8-15

## Distribute Your Add-Ins and COM Components to End Users

### Distributing MATLAB Code Using the MATLAB Runtime

On target computers without MATLAB, install the MATLAB runtime, if it is not already present on the deployment machine.

#### Install MATLAB Runtime

The *MATLAB runtime* is an execution engine made up of the same shared libraries MATLAB uses to enable execution of MATLAB files on systems without an installed version of MATLAB.

The “MATLAB runtime” is now available for downloading from the Web to simplify the distribution of your applications created using the MATLAB Compiler. Download the MATLAB runtime from the MATLAB runtime product page.

The MATLAB runtime installer does the following:

- 1 Installs the MATLAB runtime (if not already installed on the target machine)
- 2 Installs the component assembly in the folder from which the installer is run
- 3 Copies the *MWArray* assembly to the Global Assembly Cache (GAC), as part of installing the MATLAB runtime

#### MATLAB Runtime Prerequisites

- 1 Since installing the MATLAB runtime requires write access to the system registry, ensure you have administrator privileges to run the MATLAB Runtime Installer.
- 2 The version of the MATLAB runtime that runs your application on the target computer must be compatible with the version of MATLAB Compiler that built the deployed MATLAB code.
- 3 Do not install the MATLAB runtime in MATLAB installation directories.
- 4 The MATLAB runtime installer requires approximately 2 GB of disk space.

#### Add the MATLAB Runtime Installer to the Installer

This example shows how to include the MATLAB runtime in the generated installer, using one of the compiler apps. The generated installer contains all files needed to run the standalone application or shared library built with MATLAB Compiler and properly lays them out on a target system.



- 1 On the **Packaging Options** section of the compiler interface, select one or both of the following options:
  - **Runtime downloaded from web** — This option builds an installer that invokes the MATLAB runtime installer from the MathWorks Web site.
  - **Runtime included in package** — The option includes the MATLAB runtime installer into the generated installer.
- 2 Click **Package**.
- 3 Distribute the installer as needed.

### **Install the MATLAB Runtime**

This example shows how to install the MATLAB runtime on a system.

If you are given an installer containing the compiled artifacts, then the MATLAB runtime is installed along with the application or shared library. If you are given just the raw binary files, download the MATLAB runtime installer from the Web and run the installer.

---

**Note:** If you are running on a platform other than Windows, “set the system paths” on the target machine. Setting the paths enables your application to find the MATLAB runtime.

Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See “Using MATLAB Compiler on Mac or Linux” for detailed information on performing all deployment tasks specifically with UNIX variants such as Linux and Mac.

---

## Deploying Your Microsoft Visual Basic Application

### In this section...

“Calling Compiled MATLAB Functions from Microsoft Excel” on page 8-4

“Improving Data Access Using the MCR User Data Interface, COM Components, and MATLAB Builder EX” on page 8-6

“MCR Component Cache and CTF Archive Embedding” on page 8-11

“MCR Run-Time Options” on page 8-13

### 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 Create the following MATLAB functions in three separate files named `doubleit.m`, `incrementit.m`, and `powerit.m`, respectively:

```
function output = doubleit(input)
    output = input * 2;
```

```
function output = incrementit(input1, input2)
    output = input1 + input2;
```

```
function output = powerit(input1, input2)
    output = power(input1, input2);
```

- 2 Start the Library Compiler.
- 3 Use the following information as you work through this example using the instructions in “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12:

Application Name

myexcelfunctions

Class Name	myexcelfunctionsclass
Exported Functions	doubleit.m incrementit.m powerit.m

Perform the following steps on the Deployment machine:

- 1 Copy the contents of `for_redistribution_files_only` to the deployment machine(s). Copy the file to a standard place for use with Microsoft Excel, such as *Office\_Installation\_folder*\Library\MATLAB where *Office\_Installation\_folder* is a folder such as C:\Program Files\Microsoft Office\OFFICE11.
- 2 Install the MATLAB runtime.
- 3 Register `myexcelfunctions_1_0.dll`.

---

**Caution** You need to re-register your DLL file if you move it following its creation. Unlike DLL files, Excel files can be moved anywhere at anytime.

---

- 4 Start Microsoft Excel. The spreadsheet **Book1** should be open by default.
- 5 In Excel, select **Tools > Visual Basic Editor**. The Microsoft Visual Basic Editor starts.
- 6 In the Microsoft Visual Basic Editor, select **File > Import File**.
- 7 Browse to `myexcelfunctions.bas` and click **Open**. In the Project Explorer, **Module1** appears under the **Modules** node beneath **VBAProject (Book1)**.
- 8 In the Microsoft Visual Basic Editor, select **View > Microsoft Excel**. You can now use the `doubleit`, `incrementit`, and `powerit` functions in your **Book1** spreadsheet.
- 9 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 **49** in cells **A1**, **A2**, and **A3** respectively.

- 10 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\_folder\XLSTART* folder.
- d** Save the file as *Office\_Installation\_folder\XLSTART\Book.xlt*.

---

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

---

### Where Is the Example Code?

See “Example File Copying” on page 4-25 for information about accessing the example code from within the product.

## Improving Data Access Using the MCR User Data Interface, COM Components, and MATLAB Builder EX

- “Overview” on page 8-6
- “Supplying Run-Time Profile Information for Parallel Computing Toolbox Applications” on page 8-7

### Overview

This feature provides a lightweight interface for easily accessing MCR data. It allows data to be shared between an MCR instance, the MATLAB code running on that MCR, and the wrapper code that created the MCR. Through calls to the MCR User Data interface API, you access MCR data by creating a per-MCR-instance associative array of `mxArrays`, consisting of a mapping from string keys to `mxArray` values. Reasons for doing this include, but are not limited to:

- You need to supply run-time profile information to a client running an application created with the Parallel Computing Toolbox. Profile information may be supplied (and change) on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles.
- You want to initialize the MCR with constant values that can be accessed by all your MATLAB applications.
- You want to set up a global workspace — a global variable or variables that MATLAB and your client can access.
- You want to store the state of any variable or group of variables.

MATLAB Builder EX supports per-MCR instance state access through an object-oriented API. Unlike MATLAB Compiler, access to per-MCR instance state is optional, rather than on by default. You can access this state by adding `setmcruserdata.m` and `getmcruserdata.m` to your deployment project or by specifying them on the command line. Alternately, you use a helper function to call these methods, as shown in “Supplying Run-Time Profile Information for Parallel Computing Toolbox Applications” on page 8-7.

For more information, see the MATLAB Compiler User's Guide.

### Supplying Run-Time Profile Information for Parallel Computing Toolbox Applications

Following is a complete example of how you can use the MCR User Data Interface as a mechanism to specify a profile for Parallel Computing Toolbox applications.

---

**Note:** Standalone executables and shared libraries generated from MATLAB Compiler for parallel applications can now launch up to twelve local workers without MATLAB Distributed Computing Server™.

---

#### Step 1: Write Your Parallel Computing Toolbox Code

- 1 Compile `sample_pct.m` in MATLAB.

This example code uses the cluster defined in the default profile.

The output assumes that the default profile is `local`.

```
function speedup = sample_pct (n)
warning off all;
tic
if(ischar(n))
    n=str2double(n);
end
for ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time1 =toc;
parpool;
tic
parfor ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
```

```
end
time2 =toc;
disp(['Normal loop times: ' num2str(time1) ...
      ',parallel loop time: ' num2str(time2) ]);
disp(['parallel speedup: ' num2str(1/(time2/time1)) ...
      ' times faster than normal']);
delete(gcf);
disp('done');
speedup = (time1/time2);
```

- 2 Run the code as follows after changing the default profile to `local`, if needed.

```
a = sample_pct(200)
```

- 3 Verify that you get the following results;

```
Starting matlabpool using the 'local'
      profile ... connected to 4 labs.
Normal loop times: 1.4625, parallel loop time: 0.82891
parallel speedup: 1.7643 times faster than normal
Sending a stop signal to all the labs ... stopped.
done
a =
    1.7643
```

### Step 2: Set the Parallel Computing Toolbox Profile

In order to compile MATLAB code to a COM component and utilize the Parallel Computing Toolbox, the `mcruserdata` must be set directly from MATLAB. There is no API available to access the `MCRUserdata` as there is for C and C++ applications built with MATLAB Compiler.

To set the `mcruserdata` from MATLAB, create an `init` function in your COM class. This is a separate MATLAB function that uses `setmcruserdata` to set the Parallel Computing Toolbox profile once. You then call your other functions to utilize the Parallel Computing Toolbox functions.

Create the following `init` function:

```
function init_sample_pct
% Set the Parallel Profile:
if(isdeployed)
    [profile] = uigetfile('*.settings');
    % let the USER select file
    setmcruserdata('ParallelProfile',[profile]);
```

end

### Step 3: Compile Your Function with the Deployment Tool or the Command Line

You can compile your function from the command line by entering the following:

```
mcc -B 'cexcel:exPctComp,exPctClass,1.0' init_sample_pct.m sample_pct.m
```

Alternately, you can use the Deployment Tool as follows:

- 1 Follow the steps in “Create a Microsoft Excel Add-In and COM Component from MATLAB Code” on page 1-12 to compile your application.

When the compilation finishes, a new folder (with the same name as the project) is created.

Project Name	exPctComp
Class Name	exPctClass
File to compile	sample_pct.m and init_sample_pct.m

---

**Note:** If you are using the GPU feature of Parallel Computing Toolbox™, you need to manually add the PTX and CU files.

If you are using the Library Compiler app, click **Add files/directories** on the **Build** tab.

If you are using the `mcc` command, use the `-a` option.

---

- 2 To deploy the compiled application, copy the `for_redistribution_files_only` folder, which contains the following, to your end users.
  - `exPctComp.dll`
  - VBA module (`.bas` file)
  - MCR installer
  - Cluster profile

---

**Note:** The end-user's target machine must have access to the cluster.

---

**Step 4: Modify the generated VBA Driver Application (the BAS File)**

After registering the COM DLL on the deployment machine and importing the BAS file into Excel, modify the generated BAS file code as needed.

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean
Dim exPctClass As Object

Private Sub InitModule()
    If Not bModuleInitialized Then
        On Error GoTo Handle_Error
        If MCLUtil Is Nothing Then
            Set MCLUtil = CreateObject("MWComUtil.MWUtil7.10")
        End If
        Call MCLUtil.MWInitApplication(Application)
        bModuleInitialized = True
        Exit Sub
    Handle_Error:
        bModuleInitialized = False
    End If
End Sub

Function init_sample_pct() As Variant

    On Error GoTo Handle_Error
    Call InitModule
    If exPctClass Is Nothing Then
        Set exPctClass = CreateObject("exPctComp.exPctClass.1_0")
    End If
    Call exPctClass.init_sample_pct
    init_sample_pct = Empty

    Exit Function
    Handle_Error:
        init_sample_pct = "Error in " &
            Err.Source & ": " & Err.Description
End Function

Function sample_pct(Optional pelle As Variant) As Variant
    Dim speedup As Variant

    On Error GoTo Handle_Error
    Call InitModule
```



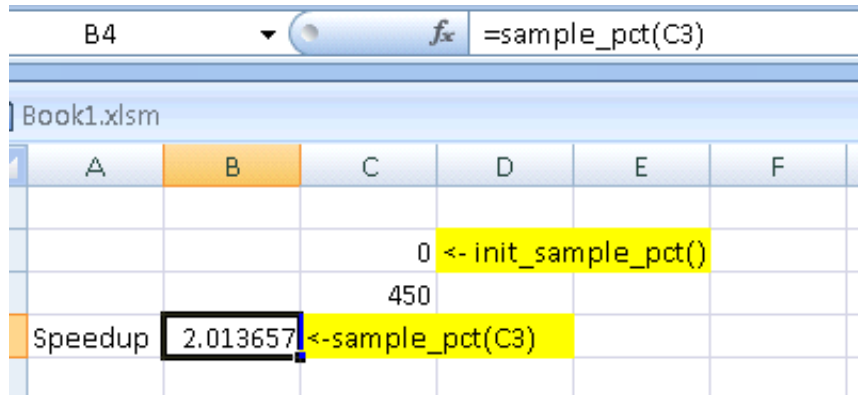
```

If exPctClass Is Nothing Then
    Set exPctClass = CreateObject("exPctComp.exPctClass.1_0")
End If
Call exPctClass.sample_pct(1, speedup, pelle)
sample_pct = speedup

Exit Function
Handle_Error:
    sample_pct = "Error in " & Err.Source
                & ": " & Err.Description
End Function

```

The output is as follows:



## MCR Component Cache and CTF Archive Embedding

- “Overriding Default Behavior” on page 8-13
- “For More Information” on page 8-13

CTF data is automatically embedded directly in MATLAB Builder EX components by default and extracted to a temporary folder.

Automatic embedding enables usage of MCR Component Cache features through environment variables.

These variables allow you to specify the following:

- Define the default location where you want the CTF archive to be automatically extracted

- Add diagnostic error printing options that can be used when automatically extracting the CTF, for troubleshooting purposes
- Tuning the MCR component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location.	Does not apply
MCR_CACHE_VERBOSE	When set to any value, this variable prints logging details about the component cache for diagnostic reasons. This can be very helpful if problems are encountered during deployable archive extraction.	Logging details are turned off by default (for example, when this variable has no value).
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mcrcachedir</code> command, with the desired cache size limit.

You can override this automatic embedding and extraction behavior by compiling with the `-C` option. See “Overriding Default Behavior” on page 8-13 for details.

---

**Note:** If you run `mcc` specifying conflicting wrapper and target types, the CTF will not be embedded into the generated component. For example, if you run:

```
mcc -W lib:myLib -T link:exe test.m test.c
```

the generated `test.exe` will not have the CTF embedded in it, as if you had specified a `-C` option to the command line.

---

---

**Caution** Do not extract the files within the `.ctf` file and place them individually under version control. Since the `.ctf` file contains interdependent MATLAB functions and data, the files within it must be accessed only by accessing the `.ctf` file. For best results, place the entire `.ctf` file under version control.

---

### Overriding Default Behavior

To extract the deployable archive in a manner prior to R2008b, alongside the compiled COM component, compile using the `mcc -c` option.

You can also implement this override by adding the `-c` flag in the **Settings** section of the compiler app.

You might want to use this option to troubleshoot problems with the deployable archive, for example, as the log and diagnostic messages are much more visible.

### For More Information

For more information about the deployable archive, see “Deployable Archive” on page 5-6.

## MCR Run-Time Options

### What MCR Options are Supported by MATLAB Builder EX?

- `-logfile` — Creates a named log file.

### How Do I Specify MCR Options?

#### If You Compiled the Add-In in MATLAB or used `mcc`

If you are building your add-in using the MATLAB Library Compiler, select **Create log file** under **Additional Runtime Settings**.

If you are building your add-in using `mcc`, simply specify `-logfile` with the `mcc -R` command

#### If You Created a Function From Scratch Using the Function Wizard

If you created a function from scratch using the Function Wizard, and want to specify MCR options, you have to manually modify the `.bas` file code.

You do this by invoking the following `MWUtil` API calls, detailed with examples in “Class `MWUtil`” on page 10-3:

- “Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])”
- “Function IsMCRJVMEEnabled() As Boolean”
- “Function IsMCRInitialized() As Boolean”

## For More Information

For more information about...	See...
Functions available through MATLAB Builder EX	Function Reference pages in MATLAB Builder EX documentation
Utility functions you can use to customize COM components created with MATLAB Builder EX	“Referencing Utility Classes” on page 10-2



# Function Reference

---

## componentinfo

Query system registry about component created with MATLAB Builder EX

### 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, <code>minor_revision_number</code> 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.
Interfaces	An array of structures defining all interface definitions in the type library. Each structure contains two fields:

Field	Description
	<ul style="list-style-type: none"> <li>• Name - Interface name.</li> <li>• IID - Interface GUID.</li> </ul>

**Registry Information Returned by componentinfo (Continued)**

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:               <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> </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

Compile and package functions for external deployment

## Syntax

```
deploytool  
deploytool project_name  
deploytool -build project_name  
deploytool -package project_name
```

## Description

`deploytool` opens the MATLAB Compiler app.

`deploytool project_name` opens the MATLAB Compiler app with the project preloaded.

`deploytool -build project_name` runs the MATLAB Compiler to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the MATLAB Compiler to build and package the specified project. The installer is generated.

## Examples

### Create a New Compiler Project

Open the compiler to create a new project.

```
deploytool
```

### Package an Application using an Existing Project

Open the compiler to build a new application using an existing project.

```
deploytool -package my_magic
```

## Input Arguments

**project\_name** — name of the project to be compiled

string

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

# libraryCompiler

Build and package functions for use in external applications

## Syntax

```
libraryCompiler  
libraryCompiler project_name  
libraryCompiler -build project_name  
libraryCompiler -package project_name
```

## Description

`libraryCompiler` opens the MATLAB shared library compiler for the creation of a new compiler project

`libraryCompiler project_name` opens the MATLAB shared library compiler app with the project preloaded.

`libraryCompiler -build project_name` runs the MATLAB shared library compiler to build the specified project. The installer is not generated.

`libraryCompiler -package project_name` runs the MATLAB shared library compiler to build and package the specified project. The installer is generated.

## Examples

### Create a New Project

Open the library compiler to create a new project.

```
libraryCompiler
```

### Package a Function using an Existing Project

Open the library compiler using an existing project.

```
libraryCompiler -package my_magic
```

## Input Arguments

**project\_name** — name of the project to be compiled  
string

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

## mcc

Compile MATLAB functions for deployment

### Syntax

```

mcc -e | -m [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-v] [-w option [:msg]] [-Y filename] mfilename
mcc -l [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -c [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -W cpplib:library_name -T link:lib [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -W dotnet:assembly_name, [className], [framework_version], security, remote_type -T link:lib [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]
mcc -W excel:addin_name, [className], [version] -T link:lib [-a filename...] [-b] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-u] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -W 'java:packageName, [className]' [-a filename...] [-b] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]
mcc -W CTF:archive_name [-a filename...] [-b] [-B filename [:arg...]] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]
mcc -W mpsxl:archive_name, [className], [version] -T link:lib [-replaceBlankWithNaN] [-convertDateToString] [-replaceNaNWithZero] [-convertNumericToDate] [-a filename...] [-b] [-B filename [:arg...]] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]

```



```

mcc -H -W mpsxl:archive_name, [className], [version] -T link:lib [-
replaceBlankWithNaN] [-convertDateToString] [-replaceNaNWithZero] [-
convertNumericToDate] [-a filename...] [-b] [-B filename [:arg...]] [-d outFolder] [-f
filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w
option [:msg]] [-Y filename] filename... [class{className: [mfilename...]}...]
mcc -H -W hadoop:archive_name,CONFIG:configFile [-a filename...] [-b] [-B
filename [:arg...]] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string]
[-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] filename...
[class{className: [mfilename...]}...]
mcc -?

```

## Description

`mcc -m mfilename` compiles the function into a standalone application.

This is equivalent to `-W main -T link:exe`.

`mcc -e mfilename` compiles the function into a standalone application that does not open an MS-DOS<sup>®</sup> command window.

This is equivalent to `-W WinMain -T link:exe`.

`mcc -l mfilename...` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This is equivalent to `-W lib:libname -T link:lib`.

`mcc -c mfilename...` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:libname -T codegen`.

`mcc -W cpplib:library_name -T link:lib mfilename...` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

`mcc -W dotnet:assembly_name,className,framework_version,security,remote_type -T link:lib mfilename...` creates a .NET assembly from the specified files.

- *assembly\_name* — Specifies the name of the assembly and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *className* — Specifies the name of the .NET class to be created.
- *framework\_version* — Specifies the version of the Microsoft .NET Framework you want to use to compile the assembly. Specify either:
  - 0.0 — Use the latest supported version on the target machine.
  - *version\_major.version\_minor* — Use a specific version of the framework.

Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.

- *security* — Specifies whether the assembly to be created is a private assembly or a shared assembly.
  - To create a private assembly, specify **Private**.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *remote\_type* — Specifies the remoting type of the assembly. Values are **remote** and **local**.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}...className` specifies the name of the class to create using *mfilename*.

`mcc -W excel:addin_name,className,version -T link:lib mfilename...` creates a Microsoft Excel add-in from the specified files.

- *addin\_name* — Specifies the name of the addin and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *addin\_name* as the default.
- *version* — Specifies the version of the add-in specified as *major.minor*.
  - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

`mcc -W 'java:packageName,className' mfilename...` creates a Java package from the specified files.

- *packageName* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *packageName*.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}...className` specifies the name of the class to create using *mfilename*.

`mcc -W CTF:archive_name` instructs the compiler to create a deployable archive that is deployable in a MATLAB Production Server instance.

`mcc -W mpsxl:addin_name,className,version input_marshall_options output_marshall_options -T link:lib mfilename...` creates an MATLAB Production Server integrated Microsoft Excel add-in from the specified files.

- *addin\_name* — Specifies the name of the add-in and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *addin\_name* as the default.
- *version* — Specifies the version of the add-in specified as *major.minor*.
  - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.
- *input\_marshall\_options* — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
  - `-replaceBlankWithNaN` — Specifies that blanks in Microsoft Excel are marshaled into MATLAB NaNs. If you do not specify this flag, blanks are marshaled as 0.
  - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB strings. If you do not specify this flag, dates are marshaled as MATLAB doubles.
- *output\_marshall\_options* — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.

- `-replaceNaNWithZero` — Specifies that MATLAB NaNs are marshaled into Microsoft Excel 0s. If you do not specify this flag, NaNs are marshaled as Visual Basic #QNANs.
- `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

`mcc -H -W hadoop:archiveName,CONFIG:configFile` generates deployable archive that can be run as a job by Hadoop®.

- *archiveName* — Specifies the name of the generated archive.
- *configFile* — Specifies the path to the Hadoop settings file. See “Hadoop Settings File”.

`mcc -?` displays help.

---

**Tip** You can issue the `mcc` command either from the MATLAB command prompt or the DOS or UNIX® command line.

---

## Options

### -a Add to Archive

Add a file to the deployable archive using

```
-a filename
```

to specify a file to be directly added to the deployable archive. Multiple `-a` options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If only a folder name is included with the `-a` option, the entire contents of that folder are added recursively to the deployable archive. For example:

```
mcc -m hello.m -a ./testdir
```

In this example, `testdir` is a folder in the current working folder. All files in `testdir`, as well as all files in subfolders of `testdir`, are added to the deployable archive, and the folder subtree in `testdir` is preserved in the deployable archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the deployable archive and subfolders of the given path are not processed recursively. For example:

```
mcc -m hello.m -a ./testdir/*
```

In this example, all files in `./testdir` are added to the deployable archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

In this example, all files with the extension `.m` under `./testdir` are added to the deployable archive and subfolders of `./testdir` are not processed recursively.

All files added to the deployable archive using `-a` (including those that match a wildcard pattern or appear under a folder specified using `-a`) that do not appear on the MATLAB path at the time of compilation causes a path entry to be added to the deployed application's run-time path so that they appear on the path when the deployed code executes.

When files are included, the absolute path for the DLL and header files is changed. The files are placed in the `.\exe_mcr\` folder when the deployable archive is expanded. The file is not placed in the local folder. This folder is created from the deployable archive the first time the application is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the `exe_mcr\exe` folder.

---

**Caution** If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

---

---

**Note:** Currently, `*` is the only supported wildcard.

---

---

**Note:** If the `-a` flag is used to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

---

## **-b Generate Excel Compatible Formula Function**

Generate a Visual Basic file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function. This option requires MATLAB Builder EX.

## **-B Specify Bundle File**

Replace the file on the `mcc` command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle file `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See “Using Bundle Files to Build MATLAB Code” for a list of the bundle files included with MATLAB Compiler.

## **-C Do Not Embed Deployable Archive by Default**

Override automatically embedding the deployable archive in C/C++ and `main/Winmain` shared libraries and standalone binaries by default.

## **-d Specified Folder for Output**

Place output in a specified folder. Use

```
-d outFolder
```

to direct the generated files to *outFolder*.

## **-e Suppress MS-DOS Command Window**

Suppress appearance of the MS-DOS command window when generating a standalone application. Use `-e` in place of the `-m` option. This option is available for Windows only. Use with `-R` option to generate error logging as such:

```
mcc -e -R -logfile -R 'filename' -v function_name
```

or:

```
mcc -e -R '-logfile,logfile' -v function_name
```

For example, to build a standalone from function `foo.m` that suppresses the MS-DOS command window, and specifying error logging to a text file, enter this command:

```
mcc -e -R '-logfile,errorlog.txt' -v foo.m
```

## **-f Specified Options File**

Override the default options file with the specified options file. Use

```
-f filename
```

to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to `mbuild`.

## **-g Generate Debugging Information**

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MATLAB runtime, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

## **-G Debug Only**

Same as `-g`.

## **-I Add Folder to Include Path**

Add a new folder path to the list of included folders. Each `-I` option adds a folder to the beginning of the list of paths to search. For example,

```
-I <directory1> -I <directory2>
```

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

## **-K Preserve Partial Output Files**

Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

## **-m Generate a Standalone Application**

Macro to produce a standalone application. This macro is equivalent to the defunct:

```
-W main -T link:exe
```

Use the `-e` option instead of the `-m` option to generate a standalone application while suppressing the appearance of the MS-DOS command window.

---

**Note:** Using the `-e` option requires the application to successfully compile with a Microsoft compiler.

---

## **-M Direct Pass Through**

Define compile-time options. Use

```
-M string
```

to pass `string` directly to `mbuild`. This provides a useful mechanism for defining compile-time options, e.g., `-M "-Dmacro=value"`.

---

**Note:** Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

---



## **-N Clear Path**

Passing `-N` effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- `matlabroot\toolbox\matlab`
- `matlabroot\toolbox\local`
- `matlabroot\toolbox\compiler\deploy`

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under `matlabroot\toolbox`.

## **-o Specify Output Name**

Specify the name of the final executable (standalone applications only). Use

```
-o outputfile
```

to name the final executable output of MATLAB Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows standalone applications).

## **-p Add Folder to Path**

Use in conjunction with the required option `-N` to add specific folders (and subfolders) under `matlabroot\toolbox` to the compilation MATLAB path in an order sensitive way. Use the syntax

```
-N -p directory
```

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included follow.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.

- If a folder is included with `-p` that is not on the original MATLAB path, that folder is not included in the compilation. (You can use `-I` to add it.)

If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the folder is added to the head of the path, as it normally would be with `-I`.

## -R Run-Time

Provides MATLAB runtime run-time options. The syntax is as follows:

`-R option`

Option	Description
<code>-logfile, filename</code>	Specify a log file name.
<code>-nodisplay</code>	Suppress the MATLAB <code>nodisplay</code> run-time warning.
<code>-nojvm</code>	Do not use the Java Virtual Machine (JVM).
<code>-startmsg</code>	Customizable user message displayed at initialization time.
<code>-completemsg</code>	Customizable user message displayed when initialization is complete.

---

**Note:** Not all `-R` options are available for all `mcc` targets.

---

---

**Caution** When running on Mac OS X, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

---

## -S Create Singleton MATLAB Runtime Context

The standard behavior for the MATLAB runtime is that every instance of a class gets its own runtime context. This runtime context includes a global MATLAB workspace for variables such as the path and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This

ensures that changes made to the global, or base, workspace in one instance of the class does not effect other instances of the same class.

In a singleton MATLAB runtime, all instances of a class share the runtime context. If multiple instances of a class are created, they use the runtime context created by the first instance. This saves start up time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all of the class instances. For example, if `instance1` creates a global variable `A` in a singleton MATLAB runtime, the `instance2` will be able to use variable `A`.

## **-T Specify Target Stage**

Specify the output target phase and type.

Use the syntax `-T target` to define the output type. Target values are as follow.

Target	Description
<code>compile:exe</code>	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a standalone application.
<code>compile:lib</code>	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a shared library/DLL.
<code>link:exe</code>	Same as <code>compile:exe</code> plus links object files into a standalone application.
<code>link:lib</code>	Same as <code>compile:lib</code> plus links object files into a shared library/DLL.

## **-u Register COM Component for the Current User**

Register COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

## **-v Verbose**

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

## **-w Warning Messages**

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings. This table lists the syntaxes.

Syntax	Description
<code>-w list</code>	Generate a table that maps <code>&lt;string&gt;</code> to warning message for use with <code>enable</code> , <code>disable</code> , and <code>error</code> . “Warning Messages”, lists the same information.
<code>-w enable</code>	Enable complete warnings.
<code>-w disable[:&lt;string&gt;]</code>	Disable specific warnings associated with <code>&lt;string&gt;</code> . “Warning Messages”, lists the <code>&lt;string&gt;</code> values. Omit the optional <code>&lt;string&gt;</code> to apply the <code>disable</code> action to all warnings.
<code>-w enable[:&lt;string&gt;]</code>	Enable specific warnings associated with <code>&lt;string&gt;</code> . “Warning Messages”, lists the <code>&lt;string&gt;</code> values. Omit the optional <code>&lt;string&gt;</code> to apply the <code>enable</code> action to all warnings.
<code>-w error[:&lt;string&gt;]</code>	Treat specific warnings associated with <code>&lt;string&gt;</code> as an error. Omit the optional <code>&lt;string&gt;</code> to apply the <code>error</code> action to all warnings.
<code>-w off[:&lt;string&gt;]</code> <code>[&lt;filename&gt;]</code>	Turn warnings off for specific error messages defined by <code>&lt;string&gt;</code> . You can also narrow scope by specifying warnings be turned off when generated by specific <code>&lt;filename&gt;</code> s.

Syntax	Description
<code>-w on[:&lt;string&gt;] [&lt;filename&gt;]</code>	Turn warnings on for specific error messages defined by <i>&lt;string&gt;</i> . You can also narrow scope by specifying warnings be turned on when generated by specific <i>&lt;filename&gt;</i> s.

It is also possible to turn warnings on or off in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using `isdeployed`) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

## **-Y License File**

Use

```
-Y license.lic
```

to override the default license file with the specified argument.

# productionServerCompiler

Build and package functions for use with MATLAB Production Server

## Syntax

```
productionServerCompiler  
productionServerCompiler project_name  
productionServerCompiler -build project_name  
productionServerCompiler -package project_name
```

## Description

`productionServerCompiler` opens the MATLAB compiler for the creation of a new compiler project

`productionServerCompiler project_name` opens the MATLAB compiler with the project preloaded.

`productionServerCompiler -build project_name` runs the MATLAB compiler to build the specified project. The installer is not generated.

`productionServerCompiler -package project_name` runs the MATLAB compiler to build and package the specified project. The installer is generated.

## Examples

### Create a New Production Server Project

Open the production server compiler to create a new project.

```
productionServerCompiler
```

### Package a Deployable Archive using an Existing Project

Open the production server compiler to package a deployable archive using an existing project file.

```
productionServerCompiler -package my_magic
```

## Input Arguments

**project\_name** — name of the project to be compiled  
string

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.





# Utility Library for Microsoft COM Components

---

- “Referencing Utility Classes” on page 10-2
- “Class MWUtil” on page 10-3
- “Class MWFlags” on page 10-12
- “Class MWStruct” on page 10-18
- “Class MWField” on page 10-25
- “Class MWComplex” on page 10-27
- “Class MWSparse” on page 10-30
- “Class MWArg” on page 10-34
- “Enum mwArrayFormat” on page 10-36
- “Enum mwDataType” on page 10-37
- “Enum mwDateFormat” on page 10-38

## Referencing Utility Classes

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

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

```
mwregsvr mwcomutil.dll
```

The `MWComUtil` library includes seven classes and three enumerated types. Before using these types, you must make explicit references to the `MWComUtil` type libraries in the Microsoft Visual Basic IDE. To do this select **Tools > References** from the main menu of the Visual Basic 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 folder in which the component resides.

---

## Class MWUtil

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

### In this section...

“Sub `MWInitApplication(pApp As Object)`” on page 10-3

“Sub `MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])`” on page 10-5

“Function `IsMCRJVMEEnabled() As Boolean`” on page 10-6

“Function `IsMCRInitialized() As Boolean`” on page 10-6

“Sub `MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])`” on page 10-7

“Sub `MWUnpack(VarArg, [nStartAt As Long], [bAutoSize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])`” on page 10-8

“Sub `MWDate2VariantDate(pVar)`” on page 10-10

The function prototypes use Visual Basic syntax.

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

Initializes the library with the current instance of Microsoft Excel.

#### Parameters

Argument	Type	Description
<code>pApp</code>	Object	A valid reference to the current Excel application

#### Return Value

None.

## Remarks

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

## Example

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

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

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

---

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

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

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

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

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

---

## Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])

Start MATLAB runtime with MATLAB runtime options. Similar to “mclInitializeApplication”.

### Parameters

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

### Return Value

None.

### Remarks

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

### Example

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

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

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

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

---

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

---

## **Function IsMCRJVMEabled() As Boolean**

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

### **Parameters**

None.

### **Return Value**

Boolean

## **Function IsMCRInitialized() As Boolean**

Returns true if MATLAB runtime is initialized; otherwise returns true

### **Parameters**

None.

### **Return Value**

Boolean

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

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

### Parameters

Argument	Type	Description
pVarArg	Variant	Receives the resulting array
[Var0], [Var1], ...	Variant	Optional list of <b>Variants</b> 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**

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



Argument	Type	Description
bAutoSize	Boolean	Optional auto-resize flag. If this flag is <b>True</b> , any Excel range output arguments are resized to fit the dimensions of the <b>Variant</b> to be copied. The resizing process is applied relative to the upper left corner of the supplied range. Default = <b>False</b> .
[pVar0],[pVar1], ...	Variant	Optional list of <b>Variants</b> to receive the array items contained in <b>VarArg</b> . 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 *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. 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("MWComUtil.MWUtil")
    Call aClass.getdates(1, R, R.Rows.Count, inc)
    Call aUtil.MWDate2VariantDate(R)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

## Class MWFlags

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

In this section...
“Property ArrayFormatFlags As MWArrayFormatFlags” on page 10-12
“Property DataConversionFlags As MWDataConversionFlags” on page 10-15
“Sub Clone(ppFlags As MWFlags)” on page 10-17

### Property ArrayFormatFlags As MWArrayFormatFlags

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

- “Property InputArrayFormat As mwArrayFormat” on page 10-12
- “Property InputArrayIndFlag As Long” on page 10-13
- “Property OutputArrayFormat As mwArrayFormat” on page 10-13
- “Property OutputArrayIndFlag As Long” on page 10-14
- “Property AutoResizeOutput As Boolean” on page 10-14
- “Property TransposeOutput As Boolean” on page 10-14

### 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”.
<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 <code>Variants</code> (the default behavior is to convert it to a cell array), the data converter converts this array to a matrix if each <code>Variant</code> 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 MATLAB Builder NE class methods. The default value is `mwArrayFormatAsIs`. The behaviors indicated by this flag are listed in the next table.

### Array Formatting Rules for Output Arrays

Value	Behavior
mwArrayFormatAsIs	Converts arrays according to the default conversion rules listed in “Data Conversion Rules”.
mwArrayFormatMatrix	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.
mwArrayFormatCell	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 10-15
- “Property `DateBias` As `Long`” on page 10-15
- “Property `InputDateFormat` As `mwDateFormat`” on page 10-16
- “Property `OutputAsDate` As `Boolean`” on page 10-17
- “Property `ReplaceMissing` As `mwReplaceMissingData`” on page 10-17

### Property `CoerceNumericToType` As `mwDataType`

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

### Property `DateBias` 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.

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.AutoSizeOutput = True
    aClass.MWFlags.ArrayFormatFlags.TransposeOutput = True
    Call aClass.myprimes(1, R, n)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

```

### Property `InputDateFormat` As `mwDateFormat`

This property converts dates passed as input parameters to method calls on .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
<code>mwDateFormatNumeric</code>	Convert dates to numeric values as indicated by the rule listed in “Data Conversion Rules”.



Value	Behavior
mwDateFormatString	Convert input dates to strings.

### PropertyOutputAsDate As Boolean

This property processes an output argument as a date. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as **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**.

### ReplaceMissing As mwReplaceMissingData

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

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

By default, the value is **mwReplaceZero**.

## Sub Clone(ppFlags As MWFlags)

Creates a copy of an **MWFlags** object.

### Parameters

Argument	Type	Description
ppFlags	MWFlags	Reference to an uninitialized <b>MWFlags</b> object that receives the copy

### Return Value

None

### Remarks

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

## Class MWStruct

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

### In this section...

“Sub `Initialize`([`varDims`], [`varFieldNames`])” on page 10-18

“Property `Item`([`i0`], [`i1`], ..., [`i31`]) As `MWField`” on page 10-19

“Property `NumberOfFields` As Long” on page 10-22

“Property `NumberOfDims` As Long” on page 10-22

“Property `Dims` As Variant” on page 10-22

“Property `FieldNames` As Variant” on page 10-22

“Sub `Clone`(`ppStruct` As `MWStruct`)” on page 10-23

### Sub `Initialize`([`varDims`], [`varFieldNames`])

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

#### Parameters

Argument	Type	Description
<code>varDims</code>	Variant	Optional array of dimensions
<code>varFieldNames</code>	Variant	Optional array of field names

#### Return Value

None.

#### Remarks

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

## Example

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

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

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

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

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

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

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

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

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

### Parameters

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

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

### Remarks

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

- Field name only

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

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

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

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

- Single index and field name

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

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

- All indices and field name

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

```

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

```

- Array of indices and field name

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

```

Dim Index(1 To 2) As Integer

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

```

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

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

```

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

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

```

The last statement resolves to

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

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

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

- Field names are case sensitive.

## Property NumberOfFields As Long

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

## Property NumberOfDims As Long

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

## Property Dims As Variant

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

## Property FieldNames As Variant

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

### Example

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

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

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

```

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

```

## Sub Clone(ppStruct As MWStruct)

Creates a copy of an MWStruct object.

### Parameters

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

### Return Value

None

### Remarks

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

### Example

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

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

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

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



## Class MWField

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

In this section...
“Property Name As String” on page 10-25
“Property Value As Variant” on page 10-25
“Property MWFlags As MWFlags” on page 10-25
“Sub Clone(ppField As MWField)” on page 10-25

### 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
<code>ppField</code>	<code>MWField</code>	Reference to an uninitialized <code>MWField</code> object to receive the copy

**Return Value**

None.

**Remarks**

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

## Class MWComplex

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

### In this section...

“Property `Real As Variant`” on page 10-27

“Property `Imag As Variant`” on page 10-27

“Property `MWFlags As MWFlags`” on page 10-28

“Sub `Clone(ppComplex As MWComplex)`” on page 10-28

### Property `Real As Variant`

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

### Property `Imag As Variant`

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

### Example

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

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

```

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

```

## Property MWFlags As MWFlags

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

## Sub Clone(ppComplex As MWComplex)

Creates a copy of an MWComplex object.

### Parameters

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

### Return Value

None

**Remarks**

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

## Class **MWSparse**

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

In this section...
“Property <code>NumRows As Long</code> ” on page 10-30
“Property <code>NumColumns As Long</code> ” on page 10-30
“Property <code>RowIndex As Variant</code> ” on page 10-30
“Property <code>ColumnIndex As Variant</code> ” on page 10-31
“Property <code>Array As Variant</code> ” on page 10-31
“Property <code>MWFlags As MWFlags</code> ” on page 10-31
“Sub <code>Clone(ppSparse As MWSparse)</code> ” on page 10-31

### Property **NumRows As Long**

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

### Property **NumColumns As Long**

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

### Property **RowIndex As Variant**

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

## Property ColumnIndex As Variant

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

## Property Array As Variant

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

## Property MWFlags As MWFlags

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

## Sub Clone(ppSparse As MWSparse)

Creates a copy of an **MWSparse** object.

### Parameters

Argument	Type	Description
ppSparse	MWSparse	Reference to an uninitialized <b>MWSparse</b> object to receive the copy

### Return Value

None.

## Remarks

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

## Example

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

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

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

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



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

## Class MWArg

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

### In this section...

“Property Value As Variant” on page 10-34

“Property MWFlags As MWFlags” on page 10-34

“Sub Clone(ppArg As MWArg)” on page 10-34

### 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 MWAArg object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

## Enum `mwArrayFormat`

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

### `mwArrayFormat` Values

Constant	Numeric Value	Description
<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	Not applicable
<code>mwTypeLogical</code>	3	<code>logical</code>
<code>mwTypeChar</code>	4	<code>char</code>
<code>mwTypeDouble</code>	6	<code>double</code>
<code>mwTypeSingle</code>	7	<code>single</code>
<code>mwTypeInt8</code>	8	<code>int8</code>
<code>mwTypeUInt8</code>	9	<code>uint8</code>
<code>mwTypeInt16</code>	10	<code>int16</code>
<code>mwTypeUInt16</code>	11	<code>uint16</code>
<code>mwTypeInt32</code>	12	<code>int32</code>
<code>mwTypeUInt32</code>	13	<code>uint32</code>

## Enum `mwDateFormat`

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

### `mwDateFormat` Values

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

# **Apps — Alphabetical List**

---

## Library Compiler

Package MATLAB programs for deployment as C/C++ shared libraries, Java packages, and .NET assemblies

### Description

The **Library Compiler** packages MATLAB functions into libraries for including MATLAB functionality in applications written in other languages.

### Open the Library Compiler

- MATLAB Toolstrip: On the **Apps** tab, under **Application Deployment**, click the app icon.
- MATLAB command prompt: Enter `libraryCompiler`.

### Examples

- “Create a C/C++ Shared Library from MATLAB Code”
- “Create a Microsoft Excel Add-In and COM Component from MATLAB Code”
- “Create a .NET Assembly From MATLAB Code”
- “Package a Deployable COM Component”
- “Create a Java Package from MATLAB Code”

### More About

- “Shared Libraries”
- “Component Creation”
- “Add-in and Component Creation”
- “Package Creation”



## Parameters

### Type

Specify the type of library into which the MATLAB code will be packaged.

**Default:** C Shared Library

### Settings

C Shared Library

Packaged code is being used to develop an application in C.

C++ Shared Library

Packaged code is being used to develop an application in C++.

Excel Add-in

Packaged code is being used as an Excel add-in.

.NET Assembly

Packaged code is being used to develop an application in .NET.

Generic COM Component

Packaged code is being used to develop an application that uses COM.

Java Package

Packaged code is being used to develop an application in Java.

### Exported Functions

Specify the MATLAB functions to package.

### Packaging Options

Specify how the MATLAB runtime is packaged with the application.

**Default:** Runtime downloaded from web

### Settings

Runtime downloaded from web

The generated installer downloads a compatible version of the MATLAB runtime, if needed.

Runtime included in package

The generated installer includes a compatible version of the MATLAB runtime.

### Settings

Specify the output folders for the packaged code.

**Default:**

- Testing folder: `for_testing`
- Redistribution folder: `for_redistribution`

### Application Information

Specify the following information:

- Splash screen
- Icon
- Version
- Name and contact information of the author
- Brief summary of the purpose
- Detailed description

You can change the default location where the application is installed and provide some notes to the installer.

The provided information is displayed as the installer runs.

### Files required for your archive to run

Specify the MATLAB files and data files that the application requires to run. The listed files are packaged into the generated archive.

**Default:** The list of files generated by the built-in dependency analysis tool.

## **Files installed with your application**

Specify additional files that are installed by the generated installer. The listed files are installed in the same folder as the installed archive.

## **Programmatic Use**

`libraryCompiler`

# Production Server Compiler

Package MATLAB programs for deployment to MATLAB Production Server

## Description

The **Production Server Compiler** packages MATLAB programs into archives for deployment to MATLAB Production Server.

## Open the Production Server Compiler

- MATLAB Toolstrip: On the **Apps** tab, under **Application Deployment**, click the app icon.
- MATLAB command prompt: Enter `productionServerCompiler`.

## Examples

- “Create a Deployable Archive for MATLAB Production Server”
- “Build Deployable Archive and Add-In”

## More About

- “Deployable Archive Creation”
- “Integrate with MATLAB Production Server”
- “MATLAB Production Server”

## Parameters

### Type

Specify the type of application the packaged code will be used in.

**Default:** Deployable Archive (.ctf)

## Settings

### Deployable Archive (.ctf)

Packaged code is being used as part of an application where the client-side portion of the application is written using one of the MATLAB Production Server APIs.

### Deployable Archive with Excel Integration

Packaged code is being used as part of an application where the client-side portion of the application is a generated Excel add-in. The app generates the Excel add-in as well.

## Exported Functions

Specify the MATLAB functions to package.

## Settings

Specify the output folders for the packaged code.

### Default:

- Testing folder: `for_testing`
- Redistribution folder: `for_redistribution`

## Archive Information

Specify the a name for the packaged application.

**Default:** Name of the first file in the **Exported Functions** list.

## Files required for your archive to run

Specify the MATLAB files and data files that the application requires to run. The listed files are packaged into the generated archive.

**Default:** The list of files generated by the built-in dependency analysis tool.

## Files installed with your application

Specify additional files that are installed by the generated installer. The listed files are installed in the same folder as the installed archive.

**Default:** The generated archive (.ctf) file and a `readme.txt` file.

### **Programmatic Use**

`productionServerCompiler`

# Data Conversion

---

## 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 a 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 `VARIANT`s 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 `VARIANT`s. `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	–	An HRESULT (signed four-byte integer representing a COM error code)
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

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual BasicType	Definition
VT_UINT	unsigned int	—	—	Unsigned integer; equivalent to type <code>unsigned int</code>
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
<anything>   VT_BYREF	—	—	—	Bitwise combine VT_BYREF with any basic type to declare as a reference to a value

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

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

#### MATLAB to COM VARIANT Conversion Rules

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
cell	A 1-by-1 cell array converts to a single	A multidimensional cell array converts	

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
	VARIANT with a type conforming to the conversion rule for the MATLAB data type of the cell contents.	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”) This object is passed as a VT_DISPATCH type.
char	A 1-by-1 char matrix converts to a VARIANT of type VT_BSTR with string length = 1.	A 1-by-L char matrix is assumed to represent a string of length L in 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_DISPATCH	VT_DISPATCH	A MATLAB sparse array is converted to an MWSparse object. (See “Class MWSparse”) This

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
			object is passed as a VT_DISPATCH type.
double	A real 1-by-1 double matrix converts to a VARIANT of type VT_R8. A complex 1-by-1 double matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional double matrix converts to a VARIANT of type VT_R8   VT_ARRAY. A complex multidimensional double matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. (See “Class MWComplex” on page 10-27)
single	A real 1-by-1 single matrix converts to a VARIANT of type VT_R4. A complex 1-by-1 single matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional single matrix converts to a VARIANT of type VT_R4   VT_ARRAY. A complex multidimensional single matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. (See “Class MWComplex” on page 10-27)
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 MATLAB functions using the MWComplex class. (See “Class MWComplex” on page 10-27)
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	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. (See “Class MWComplex” on page 10-27)

<b>MATLAB Data Type</b>	<b>VARIANT Type for Scalar Data</b>	<b>VARIANT Type for Array Data</b>	<b>Comments</b>
		to a VARIANT of type VT_DISPATCH.	
<b>int16</b>	A real 1-by-1 <b>int16</b> matrix converts to a VARIANT of type VT_I2. A complex 1-by-1 <b>int16</b> matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional <b>int16</b> matrix converts to a VARIANT of type VT_I2   VT_ARRAY. A complex multidimensional <b>int16</b> matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the <b>MWComplex</b> class. (See “Class <b>MWComplex</b> ” on page 10-27)
<b>uint16</b>	A real 1-by-1 <b>uint16</b> matrix converts to a VARIANT of type VT_UI2. A complex 1-by-1 <b>uint16</b> matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional <b>uint16</b> matrix converts to a VARIANT of type VT_UI2   VT_ARRAY. A complex multidimensional <b>uint16</b> matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the <b>MWComplex</b> class. (See “Class <b>MWComplex</b> ” on page 10-27)
<b>int32</b>	A 1-by-1 <b>int32</b> matrix converts to a VARIANT of type VT_I4. A complex 1-by-1 <b>int32</b> matrix converts to a VARIANT of type VT_DISPATCH.	A multidimensional <b>int32</b> matrix converts to a VARIANT of type VT_I4   VT_ARRAY. A complex multidimensional <b>int32</b> matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the <b>MWComplex</b> class. (See “Class <b>MWComplex</b> ” on page 10-27)
<b>uint32</b>	A 1-by-1 <b>uint32</b> matrix converts to a VARIANT of type VT_UI4. A complex 1-by-1 <b>uint32</b> matrix converts to a VARIANT of type VT_DISPATCH.	A multidimensional <b>uint32</b> matrix converts to a VARIANT of type VT_UI4   VT_ARRAY. A complex multidimensional	Complex arrays are passed to and from compiled MATLAB functions using the <b>MWComplex</b> class. (See “Class <b>MWComplex</b> ” on page 10-27)

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
		uint32 matrix converts to a VARIANT of type VT_DISPATCH.	
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_Boo1	VT_Boo1   VT_ARRAY	

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

**COM VARIANT to MATLAB Conversion Rules**

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
VT_EMPTY	Not applicable	Empty array created.
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	

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
VT_DATE	double	<p>1. 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.</p> <p>2. VARIANT dates can be optionally converted to strings. See “Data Conversion Flags” on page A-13 for more information on type coercion.</p>
VT_INT	int32	
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.</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.

<b>VARIANT Type</b>	<b>MATLAB Data Type (scalar or array data)</b>	<b>Comments</b>
<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 and COM VARIANT to MATLAB Conversion Rules. In some cases this is not possible, for example, when existing MATLAB code is used in conjunction with a third-party product like Excel.

The following table shows the array formatting flags.

### Array Formatting Flags

Flag	Description
InputArrayFormat	<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>
InputArrayIndFlag	<p>Sets the input array indirection level used with the <code>InputArrayFormat</code> 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 <code>InputArrayFormat</code> flag to the outermost</p>

<b>Flag</b>	<b>Description</b>
	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, <code>mwArrayFormatAsIs</code> , <code>mwArrayFormatMatrix</code> , and <code>mwArrayFormatCell</code> , cause the same behavior as the corresponding <code>InputArrayFormat</code> flag values.
OutputArrayIndFlag	(Applies to nested cell arrays only.) Output array indirection level used with the <code>OutputArrayFormat</code> flag. This flag works exactly like <code>InputArrayIndFlag</code> .
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 <code>True</code> to resize each Excel range to fit the output array.
TransposeOutput	Set this flag to <code>True</code> to transpose the output arguments. Useful when calling a 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-13

“InputDateFormat” on page A-14

“OutputAsDate As Boolean” on page A-15

“DateBias As Long” on page A-15

### 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`. 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.



# Troubleshooting

---

## Errors and Solutions

In this section...
“MATLAB Builder EX Errors and Suggested Solutions” on page B-2
“Required Locations to Develop and Use Components” on page B-4
“Microsoft Excel Errors and Suggested Solutions” on page B-5
“Function Wizard Problems” on page B-6

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

**Note:** MATLAB Builder EX uses MATLAB Compiler to generate components. This means that you might see diagnostic messages from MATLAB Compiler. 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 -client mbuild_com</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 folders to <code>matlabroot\bin\win32</code> ( <code>matlabroot</code> represents the location of MATLAB on your system), and run the command <code>mwregsvr mwcomutil.dll</code> .  See “Add-in and COM Component Registration” on page 1-17 for full details.
Error in VBAProject: ActiveX component can't create object.	<ul style="list-style-type: none"> <li>Project DLL is not registered.</li> </ul>	If the DLL is not registered, open a DOS window, change folders to <code>&lt;projectdir&gt;\distrib</code>



Message	Probable Cause	Suggested Solution
	<ul style="list-style-type: none"> <li>An incompatible MATLAB DLL exists somewhere on the system path.</li> </ul>	<p>(&lt;projectdir&gt; represents the location of your project files), and run the command:  <code>mwregsvr &lt;projectdll&gt;.dll.</code></p> <p>See “Add-in and COM Component Registration” on page 1-17 for full details.</p>
<p>Error in VBAProject: Automation error The specified module could not be found.</p>	<p>This usually occurs if MATLAB is not on the system path. This error message occurs if you have more than one version of MATLAB on your system path.</p>	<p>Anytime you have multiple versions of MATLAB, ensure that the newest version of MATLAB appears on your path first. You can verify that the newest version of MATLAB is on the path first by typing <code>path</code> at the DOS prompt. See the table “Required Locations to Develop and Use Components” on page B-4.</p>
<p>LoadLibrary ("component_name.dll") failed - The specified module could not be found.</p>	<p>You may get this error message while registering the project DLL from the DOS prompt. This usually occurs if MATLAB is not on the system path.</p>	<p>See the table “Required Locations to Develop and Use Components” on page B-4.</p>
<p>Cannot recompile the M file xxxx because it is already in the library libmmfile.mlib.</p>	<p>The name you have chosen for your MATLAB file duplicates the name of a MATLAB file already in the library of precompiled MATLAB files.</p>	<p>Rename the MATLAB file, choosing a name that does not duplicate the name of a MATLAB file already in the library of precompiled MATLAB files.</p>
<p>Arguments may only be defaulted at the end of an argument list.</p>	<p>You have modified the VB script generated for MATLAB Builder EX and have not provided one or more arguments used in the modified script.</p>	<p>Provide a value for any argument that requires an explicit value. Arguments that accept defaults appear at the end of the argument list.</p>


Message	Probable Cause	Suggested Solution
Unable to use accessibility screen-readers or assistive technologies, such as JAWS <sup>®</sup> ,	Required files <code>JavaAccessBridge.dll</code> and <code>WindowsAccessBridge.dll</code> no longer added automatically to your Windows path.	Add the following DLLs to your Windows path: <code>JavaAccessBridge.dll</code> <code>WindowsAccessBridge.dll</code>
Error in <code>class.method.version</code>	This is a generic message, but is sometimes seen when there are conflicts in architecture versions of MATLAB and Microsoft Excel.	See “Deployment Target Architectures and Compatibility” on page 1-8 for detailed information.
Error: Error IMWDispatchDriver "Method Load of object IMWDispatchDriver failed"	Different versions of MCR and MATLAB results in the IMWDispatchDriver error.	Check for the current versions of MATLAB and MCR. Verify the MATLAB version by typing the MATLAB path in DOS prompt. If the versions do not match update and install the new version of MCR that matches the current MATLAB version.
Error in VBA project: Method xxx of object 'IClass1' failed	Multiple versions of MATLAB running on the system results in this error.	Register the <code>mwcomutil.dll</code> and <code>mwcommgr.dll</code> registry files. Open a DOS window, cd to <code>matlabroot\bin\win64</code> or <code>win32</code> ( <code>matlabroot</code> represents the location of MATLAB on your system), and run the command <code>mwregsvr mwcomutil.dll</code> <code>mwregsvr mwcommgr.dll</code>

## Required Locations to Develop and Use Components

Component	Development Machine	Target Machine
MCR	Make sure that <code>matlabroot\bin\win32</code> appears on your system path	Verify that <code>mcr_root\ver\runtime\win32</code> appears on your system path. ( <code>mcr_root</code> is your root MCR folder.)

Component	Development Machine	Target Machine
	ahead of any other MATLAB installations. ( <i>matlabroot</i> is your root MATLAB folder.)	


## 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 <b>Medium</b> on the <b>Security Level</b> tab by doing the following:</p> <ul style="list-style-type: none"> <li>• For Microsoft Office 2003: <ol style="list-style-type: none"> <li><b>1</b> Click <b>Tools &gt; Macro &gt; Security</b>.</li> <li><b>2</b> For <b>Security Level</b>, select <b>Medium</b>.</li> </ol> </li> <li>• For Microsoft Office 2007: <ol style="list-style-type: none"> <li><b>1</b> Click the 2007 Office button on the Microsoft Office ribbon (  ).</li> <li><b>2</b> Click <b>Excel Options &gt; Trust Center &gt; Trust Center Settings &gt; Macro Settings</b>.</li> <li><b>3</b> In <b>Developer Macro Settings</b>, select <b>Trust access to the VBA project object model</b>.</li> </ol> </li> <li>• For Microsoft Office 2010:</li> </ul>

Message	Probable Cause	Suggested Solution
		<ol style="list-style-type: none"> <li><b>1</b> Click <b>File &gt; Options &gt; Trust Center &gt; Trust Center Settings &gt; Macro Settings</b>.</li> <li><b>2</b> In <b>Developer Macro Settings</b>, select <b>Trust access to the VBA project object model</b>.</li> </ol>

## Function Wizard Problems

Problem	Probable Cause	Suggested Solution
The Function Wizard Help does not appear.	The Function Wizard Help file (mlfunction.chm) is not in the same folder as the Function Wizard add-in (mlfunction.xla).	Copy the Help file (mlfunction.chm) into the same folder as the add-in.
<p>The Function Wizard did not automatically import your .bas file, and you have to create your macro manually</p> <p>You get an error when trying to create a macro with the Function Wizard</p>	The Function Wizard has malfunctioned with an unspecified error	<ol style="list-style-type: none"> <li><b>1</b> Open Excel</li> <li><b>2</b> Do one of the following: <ul style="list-style-type: none"> <li>• If you use Microsoft Office 2007 or 2010, click <b>Developer &gt; Macros</b>.</li> <li>• If you use Microsoft Office 2003, click <b>Tools &gt; Macros &gt; Macro</b>.</li> </ul> </li> </ol> <hr/> <p><b>Tip</b> You may need to enable the <b>Developer</b> menu item before performing this step. To do this:</p>

Problem	Probable Cause	Suggested Solution
		<p><b>a</b> Click the 2007 Office button on the Microsoft Office</p>  <p>ribbon ( ) or, in Office 2010, click <b>File</b> to display the Office Backstage View.</p> <p><b>b</b> Click <b>Excel Options</b>.</p> <p><b>c</b> In the <b>Top Options for Working With Excel</b> area, select <b>Show Developer tab in the Ribbon</b>.</p> <hr/> <p><b>3</b> From the Visual Basic Editor, select <b>File &gt; Import</b> and select the created VBA file from the &lt;project_dir&gt;\distrib folder.</p>
<p>The message <b>Failed to start MATLAB</b> appears instead of <b>Starting MATLAB...</b> when MATLAB is invoked by the Function Wizard.</p>	<p>This message may appear if you manually terminate the MATLAB session that is invoked from the Function Wizard. As a result, you can no longer use the wizard's MATLAB-related features in your current Excel session.</p>	<p>Save your work and restart Microsoft Excel.</p>

<b>Problem</b>	<b>Probable Cause</b>	<b>Suggested Solution</b>
When I use CTRL + arrow keys to select ranges with the Function Wizard, once I select a function and begin to select the function inputs, keyboard navigation no longer works in excel.	This behavior results from a bug in Microsoft Excel.	If you must use arrow keys to select ranges, apply the following fix from the Microsoft Web site: <a href="http://support.microsoft.com/kb/291110">http://support.microsoft.com/kb/291110</a> .

## Deployment Product Terms

### A

*Add-in* — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic.

*Application program interface (API)* — A set of classes, methods, and interfaces that is used to develop software applications. Typically an API is used to provide access to specific functionality. See *MWArray*.

*Application* — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

*Assembly* — An executable bundle of code, especially in .NET.

### B

*Binary* — See *Executable*.

*Boxed Types* — Data types used to wrap opaque C structures.

*Build* — See *Compile*.

### C

*Class* — A user-defined type used in C++, C#, and Java, among other object-oriented languages, that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes relate in a class hierarchy. One class is a specialization (a *subclass*) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a *MATLAB class*

*Compile* — In MATLAB Compiler terminology, to compile MATLAB code involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code is compiled into a Java

package, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

*COM component* — In MATLAB Builder EX, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Builder NE, an executable component, to be integrated with Microsoft COM applications.

*Console application* — Any application that is executed from a system command prompt window.

## **D**

*Data Marshaling* — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the `MWArray` API—must be performed manually, often at great cost.

*Deploy* — The act of integrating MATLAB code into a larger-scale computing environment, usually to an enterprise application, and often to end users.

*Deployable archive* — The deployable archive is embedded by default in each generated binary by MATLAB Compiler. It houses the deployable package. All MATLAB-based content in the deployable archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details” in the MATLAB Compiler documentation.

*DLL* — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

## **E**

*Empties* — Arrays of zero (0) dimensions.

*Executable* — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

## **F**

*Fields* — For this definition in the context of MATLAB Data Structures, see *Structs*.

*Fields and Properties* — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.



## I

*Integration* — Combining deployed MATLAB code's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment.

*Instance* — For the definition of this term in context of MATLAB Production Server software, see *MATLAB Production Server Server Instance*.

## J

*JAR* — Java archive. In computing software, a JAR file (or Java ARchive) aggregates many files into one. Software developers use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

*Java-MATLAB Interface* — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

*JDK* — The *Java Development Kit* is a free Oracle® product which provides the environment required for programming in Java.

*JMI Interface* — see *Java-MATLAB Interface*.

*JRE* — *Java Run-Time Environment* is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

## M

*Magic Square* — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

*MATLAB Runtime* — An execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB.

*MATLAB Runtime Singleton* — See *Shared MATLAB Runtime Instance*.

*MATLAB Runtime Workers* — A MATLAB runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MATLAB runtime session, using the `--num-workers` options in the server configurations file.

*MATLAB Production Server Client* — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

*MATLAB Production Server Configuration* — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, “`main_config`”.

*MATLAB Production Server Server Instance* — A logical server configuration created using the `mps -new` command in MATLAB Production Server software.

*MATLAB Production Server Software* — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, Web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

*Marshaling* — See *Data Marshaling*.

*mbuild* — MATLAB Compiler command that compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

*mcc* — The MATLAB command that invokes MATLAB Compiler. It is the command-line equivalent of using the compiler apps.

*Method Attribute* — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

*mxArray interface* — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

*MWArray interface* — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`. There are different implementations of the `MWArray` proxy for each application programming language.

## P

*Package* — The act of bundling the deployed MATLAB code, along with the MATLAB runtime and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of loose artifacts that can be used for testing or building a custom installer.

*PID File* — See *Process Identification File (PID File)*.

*Pool* — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a *pool*, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

*Process Identification File (PID File)* — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

*Program* — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

*Properties* — For this definition in the context of .NET, see *Fields and Properties*.

*Proxy* — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

## S

*Server Instance* — See MATLAB Production Server Server Instance.

*Shared Library* — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept in for Microsoft Windows.

*Shared MATLAB Runtime Instance* — When using MATLAB Builder NE or MATLAB Builder JA, you can create a shared MATLAB runtime instance, also known as a *singleton*. For builder NE, this only applies to COM components. When you invoke MATLAB Compiler with the `-S` option through the builders (using either `mcc` or a compiler app), a single MATLAB runtime instance is created for each COM component or Java package in an application. You reuse this instance by sharing it among all subsequent class instances. Such sharing results in more efficient memory usage and eliminates the MATLAB runtime startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the deployed MATLAB files. MATLAB Builder NE and MATLAB Builder EX are designed to create singletons by default for .NET assemblies and COM components, respectively. For more information, see “Sharing a MATLAB Runtime Instance in COM or Java Applications”.

*State* — The present condition of a the MATLAB, or MATLAB runtime, runtime. MATLAB functions often carry state in the form of variable values. The MATLAB workspace itself also maintains information about global variables and path settings. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions.

*Structs* — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

*System Compiler* — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio.

## **T**

*Thread* — A portion of a program that can run independently of and concurrently with other portions of the program. See *pool* for additional information on managing the number of processing threads available to a server instance.

*Type-safe interface* — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application.

## **W**

*Web Application Archive (WAR)* —In computing, a Web Application Archive is a JAR file used to distribute a collection of `JavaServer` pages, servlets, Java classes, XML files, tag libraries, and static Web pages (HTML and related files) that together constitute a Web application.

*Webfigure* — A MathWorks representation of a MATLAB figure, rendered on the Web. Using the WebFigures feature, you display MATLAB figures on a Web site for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the Web, without the need to download MATLAB or other tools that can consume costly resources.

*Windows Communication Foundation (WCF)* — The Windows Communication Foundation™ is an application programming interface in the .NET Framework for building connected, service-oriented, Web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed by client applications.

