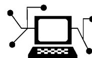
# MATLAB® Builder™ NE 3
User's Guide

MATLAB®

**How to Contact MathWorks**



| | |
|---|---|
| www.mathworks.com | Web |
| comp.soft-sys.matlab | Newsgroup |
| www.mathworks.com/contact_TS.html | Technical Support |

| | |
|---|---|
| suggest@mathworks.com | Product enhancement suggestions |
| bugs@mathworks.com | Bug reports |
| doc@mathworks.com | Documentation error reports |
| service@mathworks.com | Order status, license renewals, passcodes |
| info@mathworks.com | Sales, pricing, and general information |

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

*MATLAB® Builder™ NE User's Guide*

© COPYRIGHT 2002–2010 by The MathWorks, Inc.

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

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

**Trademarks**

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

**Patents**

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

**Revision History**

| | | |
|---|---|---|
| March 2006 | Online only | New for Version 2.0 (Release 2006a) |
| September 2006 | Online only | Revised for Version 2.1 (Release 2006b) |
| March 2007 | Online only | Revised for Version 2.2 (Release 2007a) |
| September 2007 | Online only | Revised for Version 2.2.1 (Release 2007b) |
| March 2008 | Online only | Revised for Version 2.2.2 (Release 2008a) |
| October 2008 | Online only | Revised for Version 3.0 (Release 2008b) |
| March 2009 | Online only | Revised for Version 3.0.1 (Release 2009a) |
| September 2009 | Online only | Revised for Version 3.0.2 (Release 2009b) |
| March 2010 | Online only | Revised for Version 3.1 (Release 2010a) |
| September 2010 | Online only | Revised for Version 3.2 (Release 2010b) |

# Contents

## 1 Getting Started

**Product Overview** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-2**
MATLAB® Compiler Extension . . . . . . . . . . . . . . . . . . . . . . **1-2**

**Before You Use MATLAB® Builder NE** . . . . . . . . . . . . . . . **1-3**
Your Role in the .NET Application Deployment Process . . **1-3**
What You Need to Know . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-4**
Install Products, Compilers, and IDEs . . . . . . . . . . . . . . . . **1-5**
Install Microsoft .NET Framework . . . . . . . . . . . . . . . . . . . **1-5**
Consider Architecture of Deployment Targets . . . . . . . . . . **1-6**
Evaluate Dependencies and Non-Compilable Code . . . . . . **1-6**
For More Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-6**
Troubleshooting Installation Problems . . . . . . . . . . . . . . . . **1-7**

**Deploying a Component Using the Magic Square Example** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-8**
About This Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-8**

**Magic Square Example: MATLAB Programmer Tasks** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-10**
Copying the Example Files . . . . . . . . . . . . . . . . . . . . . . . . . **1-11**
Testing the MATLAB File You Want to Deploy . . . . . . . . . **1-11**
Building Your Component . . . . . . . . . . . . . . . . . . . . . . . . . **1-13**
Packaging Your Component (Optional) . . . . . . . . . . . . . . . . **1-15**
Copying the Package You Created . . . . . . . . . . . . . . . . . . . **1-19**

**Magic Square Example: .NET Developer Tasks** . . . . . . . **1-20**
Gathering Files Needed for Deployment . . . . . . . . . . . . . . . **1-21**
Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-21**
Integrating Your Component into a .NET Class Using Microsoft® Visual Studio . . . . . . . . . . . . . . . . . . . . . . . . . **1-25**
Building and Testing the .NET Application with Microsoft® Visual Studio . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-34**

**Using the Magic Square Component in an Application** ........ **1-36**

**Next Steps** ........ **1-39**

## 2 Writing Deployable MATLAB Code

**The MATLAB Application Deployment Products** ..... **2-2**

**How the Deployment Products Process MATLAB Function Signatures** ........ **2-4**
The MATLAB Function Signature ........ **2-4**
MATLAB Programming Basics ........ **2-5**
Returning MATLAB Data Types ........ **2-5**

**The Application Deployment Products and the Deployment Tool** ........ **2-8**
What Is the Difference Between the Deployment Tool and the mcc Command Line? ........ **2-8**
How Does MATLAB® Compiler Software Build My Application? ........ **2-9**
What You Should Know About the Dependency Analysis Function (depfun) ........ **2-10**
Compiling MEX-Files, DLLs, or Shared Libraries ....... **2-10**
The Role of the Component Technology File (CTF Archive) ........ **2-11**

**Guidelines for Writing Deployable MATLAB Code** .... **2-15**
Compiled Applications Do Not Process MATLAB Files at Runtime ........ **2-15**
Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files ........ **2-16**
Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths ........ **2-16**
Gradually Refactor Applications That Depend on Noncompilable Functions ........ **2-17**
Do Not Create or Use Nonconstant Static State Variables ........ **2-17**

**Working with MATLAB Data Files Using Load and Save** . . . . . . **2-19**
Using Load/Save Functions to Process MATLAB Data for Deployed Applications . . . . . . **2-19**

## 3 Building Your .NET Component

**Supported Compilation Targets** . . . . . . **3-2**
.NET Component . . . . . . **3-2**
COM Components . . . . . . **3-3**

**Using the Deployment Tool GUI to Build .NET Components** . . . . . . **3-4**

**Using the Command Line to Build .NET Components** . . **3-5**
Command-Line Syntax Description . . . . . . **3-5**
Using the Command Line to Start the Deployment Tool GUI . . . . . . **3-7**

**For More Information** . . . . . . **3-8**

## 4 Integrating Your .NET Component

**Overview of Basic Integration Tasks** . . . . . . **4-2**
Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR) . . . . . . **4-3**

**Coding Your .NET Application** . . . . . . **4-7**
About Your .NET Application and C# Code . . . . . . **4-7**
Perform Data Conversion Where Required . . . . . . **4-9**
Using MATLAB API Functions in a C# Program . . . . . . **4-20**

Accessing Real or Imaginary Components Within Complex Arrays . . . 4-22
Adding Fields to Data Structures and Data Structure Arrays . . . 4-24
Using MATLAB Array Indexing . . . 4-24
Blocking Execution of a Console Application that Creates Figures . . . 4-25
Handling Errors . . . 4-27
Using Dispose to Explicitly Free Resources . . . 4-28

**Deploying Your .NET Component to End Users** . . . 4-30
Dynamically Specifying Run-Time Options to the MCR . . 4-30
Extracting the CTF Archive Manually Using the MCR Component Cache . . . 4-32
Improving Data Access Using the MCR User Data Interface and MATLAB® Builder NE . . . 4-33
Using Enhanced XML Documentation Files . . . 4-38

**Accessing Your Component On Another Computer** . . . 4-40

**For More Information** . . . 4-41

## 5 Sample Applications (C#)

**Simple Plot Example** . . . 5-2
Purpose . . . 5-2
Procedure . . . 5-2

**Passing Variable Arguments** . . . 5-7

**Spectral Analysis Example** . . . 5-13
Purpose . . . 5-13
Procedure . . . 5-15

**Matrix Math Example** . . . 5-20
Purpose . . . 5-20
Procedure . . . 5-21

MATLAB Functions to Be Encapsulated . . . . . . . . . . . . . . . 5-26
Understanding the MatrixMath Program . . . . . . . . . . . . . . 5-27

**Phonebook Example** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-28**
Purpose . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-28
Procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-28

## 6 Sample Applications (Microsoft® Visual Basic .NET)

**Magic Square Example (Visual Basic)** . . . . . . . . . . . . . . . **6-3**

**Create Plot Example (Visual Basic)** . . . . . . . . . . . . . . . . . . **6-7**

**Variable Arguments Example (Visual Basic)** . . . . . . . . . **6-11**

**Spectral Analysis Example (Visual Basic)** . . . . . . . . . . . . **6-15**

**Matrix Math Example (Visual Basic)** . . . . . . . . . . . . . . . . . **6-20**

**Phonebook Example (Visual Basic)** . . . . . . . . . . . . . . . . . . **6-25**
makephone Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-25
Procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-25

## 7 Deploying a MATLAB Figure Over the Web Using WebFigures

**About the WebFigures Feature** . . . . . . . . . . . . . . . . . . . . . . **7-2**
Supported Renderers for WebFigures . . . . . . . . . . . . . . . . . 7-2

**Before You Use WebFigures** . . . . . . . . . . . . . . . . . . . . . . . **7-3**

Your Role in the .NET WebFigure Deployment Process ... 7-3
What You Need to Know to Implement WebFigures ..... 7-5
Required Products ..... 7-5
Assumptions About the Examples ..... 7-6

**Quick Start: Implementing a WebFigure** ..... **7-7**
Overview ..... 7-7
Procedure ..... 7-7

**Advanced Configuration of a WebFigure** ..... **7-15**
Overview ..... 7-15
Manually Installing WebFigureService ..... 7-17
Retrieving Multiple WebFigures From a Component ..... 7-18
Attaching a WebFigure ..... 7-21
Setting Up WebFigureControl for Remote Invocation ..... 7-23
Getting an Embeddable String That References a WebFigure Attached to a WebFigureService ..... 7-25
Improving Processing Times for JavaScript Using Minification ..... 7-27
Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up ..... 7-28

**Upgrading Your WebFigures** ..... **7-30**

**Troubleshooting** ..... **7-31**

**Logging Levels** ..... **7-33**

## 8 Working with MATLAB Figures and Images

**Your Role in Working with Figures and Images** ..... **8-2**

**Creating and Modifying a MATLAB Figure** ..... **8-3**
Preparing a MATLAB Figure for Export ..... 8-3
Changing the Figure (Optional) ..... 8-3
Exporting the Figure ..... 8-4
Cleaning Up the Figure Window ..... 8-4

Example: Modifying and Exporting Figure Data ........ **8-5**

**Working with MATLAB Figure and Image Data** ...... **8-6**
For More Comprehensive Examples .................. **8-6**
Working with Figures ............................. **8-6**
Working with Images .............................. **8-7**

## 9 Sharing Components Across Distributed Applications Using .NET Remoting

**Overview** ......................................... **9-2**
What Are Remotable Components? ................... **9-2**
Benefits of Using .NET Remoting .................... **9-2**

**Your Role in Building Distributed Applications** ...... **9-4**

**Selecting the Best Method of Accessing Your Component: MWArray API or Native .NET API** ..... **9-5**
Using Native .NET Structure and Cell Arrays .......... **9-6**

**Creating a Remotable .NET Component** ............. **9-7**
Building a Remotable Component Using the Deployment Tool ......................................... **9-7**
Building a Remotable Component Using the mcc Command ...................................... **9-10**
Files Generated by the Compilation Process ............ **9-11**

**Enabling Access to a Remotable .NET Component** .... **9-12**
Using the MWArray API ........................... **9-12**
Using the Native .NET API: The Magic Square Example .. **9-19**
Using the Native .NET API: The Cell and Struct Example ....................................... **9-27**

## 10 Troubleshooting

**Troubleshooting the Build Process** ................. **10-2**
Viewing the Latest Build Log ......................... **10-2**
Generating Verbose Output ......................... **10-2**

**Failure to Find a Required File** ..................... **10-3**

**Diagnostic Messages** ............................... **10-4**
Enhanced Error Diagnostics Using mstack Trace ....... **10-7**

## 11 Reference Information

**Requirements for the MATLAB® Builder NE Product** .. **11-2**
System Requirements ............................ **11-2**
Compiler Requirements ........................... **11-2**
Path Modifications Required for Accessibility ........... **11-2**
Limitations and Restrictions ....................... **11-3**

**Data Conversion Rules** ............................ **11-4**
Managed Types to MATLAB Arrays ................. **11-4**
MATLAB Arrays to Managed Types ................. **11-5**
Character and String Conversion ................... **11-5**
Unsupported MATLAB Array Types ................. **11-6**

**Overview of Data Conversion Classes** ............... **11-7**
Overview ........................................ **11-7**
Returning Data from MATLAB to Managed Code ....... **11-8**
Example of MWNumericArray in a .NET Application .... **11-8**
Interfaces Generated by the MATLAB® Builder NE Product ....................................... **11-8**

**MWArray Class Specification** ...................... **11-14**

## 12 Function Reference

## 13 Creating and Installing COM Components

**Building a Deployable COM Component** ............. **13-2**

**Packaging a Deployable COM Component** ........... **13-4**

**About Embedded CTF Archives** ...................... **13-5**

**Using the Command-Line Interface** ................. **13-6**

**Installing COM Components on a Target Computer** ... **13-9**

## 14 Programming with COM Components Created by the MATLAB® Builder NE Product

**General Techniques** ............................... **14-2**

**Registering and Referencing the Utility Library** ...... **14-4**

**Creating an Instance of a Class in Microsoft® Visual Basic** ........................................ **14-5**
Advantages and Disadvantages ...................... **14-5**
CreateObject Function ............................. **14-5**
Microsoft® Visual Basic New Operator ................ **14-6**
Advantages of Each Technique ...................... **14-7**
Declaring a Reusable Class Instance .................. **14-7**

**Calling the Methods of a Class Instance** .............. **14-8**

Standard Mapping Technique ..... 14-8
Variant ..... 14-9
Examples of Passing Input and Output Parameters ..... 14-9

**Calling a COM Object in a Visual C++ Program** ..... **14-11**
Using the MATLAB® Builder NE Product to Create the Object ..... 14-11
Using the Component in a Visual C++ Program ..... 14-12

**Using a COM Component in a .NET Application** ..... **14-14**
Overview ..... 14-14
C# Implementation ..... 14-14
Microsoft® Visual Basic Implementation ..... 14-17

**Adding Events to COM Objects** ..... **14-21**
MATLAB Language Pragma ..... 14-21
Using a Callback with a Microsoft® Visual Basic Event ..... 14-22

**Passing Arguments** ..... **14-26**
Overview ..... 14-26
Creating and Using a varargin Array in Microsoft® Visual Basic Programs ..... 14-26
Creating and Using varargout in Microsoft® Visual Basic Programs ..... 14-27
Passing an Empty varargin From Microsoft® Visual Basic Code ..... 14-28

**Using Flags to Control Array Formatting and Data Conversion** ..... **14-29**
Overview ..... 14-29
Array Formatting Flags ..... 14-30
Using Array Formatting Flags ..... 14-30
Using Data Conversion Flags ..... 14-33
Special Flags for Some Microsoft® Visual Basic Types ..... 14-35

**Using MATLAB Global Variables in Microsoft® Visual Basic** ..... **14-36**

**Blocking Execution of a Console Application that Creates Figures** ..... **14-39**
MCRWaitForFigures ..... 14-39

Using MCRWaitForFigures to Block Execution ......... 14-40

**Obtaining Registry Information** ..................... **14-42**

**Handling Errors During a Method Call** .............. **14-44**

# 15 Using COM Components in Microsoft® Visual Basic Applications

**Magic Square Example** ............................ **15-2**
Example Overview ................................ 15-2
Creating the MATLAB File ......................... 15-2
Using the Deployment Tool to Create and Build the Project ........................................ 15-3
Creating the Microsoft® Visual Basic Project ........... 15-3
Creating the User Interface ......................... 15-4
Creating the Executable in Microsoft® Visual Basic ...... 15-7
Testing the Application ............................ 15-7
Packaging the Component .......................... 15-8

**Creating an Excel Add-In: Spectral Analysis Example** ........................................ **15-10**
Example Overview ................................ 15-10
Building the Component ........................... 15-10
Integrating the Component with VBA ................ 15-12
Creating the Microsoft® Visual Basic Form ............. 15-14
Adding the Spectral Analysis Menu Item to Microsoft® Excel ........................................ 15-20
Saving the Add-In ................................ 15-21
Testing the Add-in ................................ 15-21
Packaging and Distributing the Add-In ............... 15-23

**Univariate Interpolation Example** ................. **15-25**
Example Overview ................................ 15-25
Using the Deployment Tool to Create and Build the Component ...................................... 15-25
Using the Component in Microsoft® Visual Basic ........ 15-26

Creating the Microsoft® Visual Basic Form ............. **15-27**

**Matrix Calculator Example** ......................... **15-33**
Example Overview ................................ **15-33**
Building the Component .......................... **15-33**
Using the Component in Microsoft® Visual Basic ........ **15-34**
Creating the Microsoft® Visual Basic Form ............. **15-35**

**Curve Fitting Example** ............................ **15-44**
Example Overview ................................ **15-44**
Building the Component .......................... **15-44**
Building the Project ............................ **15-45**
Using the Component in Microsoft® Visual Basic ........ **15-45**
Creating the Microsoft® Visual Basic Form ............. **15-45**

**Bouncing Ball Simulation Example** .................. **15-52**
Example Overview ................................ **15-52**
Building the Component .......................... **15-52**
Using the Component in Microsoft® Visual Basic ........ **15-53**
Creating the Microsoft® Visual Basic Form ............. **15-54**

## 16 How the MATLAB® Builder NE Product Creates COM Components

**Overview of Internal Processes** ..................... **16-2**
How Is a MATLAB® Builder NE Component Created? ... **16-2**
Code Generation ................................. **16-2**
Create Interface Definitions ........................ **16-3**
C++ Compilation ................................. **16-3**
Linking and Resource Binding ...................... **16-3**
Registration of the DLL ........................... **16-3**

**Component Registration** ........................... **16-4**
Self-Registering Components ....................... **16-4**
Globally Unique Identifier .......................... **16-5**
Versioning ...................................... **16-7**

**Data Conversion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **16-9**
Conversion Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **16-9**
Array Formatting Flags . . . . . . . . . . . . . . . . . . . . . . . . . . **16-19**
Data Conversion Flags . . . . . . . . . . . . . . . . . . . . . . . . . . . **16-21**

**Calling Conventions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **16-23**
Producing a COM Class . . . . . . . . . . . . . . . . . . . . . . . . . . **16-23**
IDL Mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **16-24**
Microsoft® Visual Basic Mapping . . . . . . . . . . . . . . . . . . . **16-25**

## 17 Utility Library for Microsoft COM Components

**Referencing Utility Classes** . . . . . . . . . . . . . . . . . . . . . . . . **17-2**

**Utility Library Classes** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-3**
Class MWUtil . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-3**
Class MWFlags . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-10**
Class MWStruct . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-16**
Class MWField . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-23**
Class MWComplex . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-24**
Class MWSparse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-26**
Class MWArg . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-29**

**Enumerations** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-31**
Enum mwArrayFormat . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-31**
Enum mwDataType . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-31**
Enum mwDateFormat . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17-32**

## A Examples

**Magic Square Example for .NET** . . . . . . . . . . . . . . . . . . . **A-2**

**Using Load and Save** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **A-2**

**Native Data Conversion** .......... A-2

**Using functions engOpen and engEvalString from the MATLAB Engine API in a C# Program** .......... A-2

**Using WaitForFiguresToDie to Block Execution** .......... A-2

**Using the MCR Data Interface** .......... A-2

**Sample Applications (C#)** .......... A-3

**Sample Applications (Visual Basic .NET)** .......... A-3

**Quick Start to Implementing a WebFigure** .......... A-3

**Working with Functions that Return a Single WebFigure as the Function's Only Output** .......... A-3

**Working With Functions That Return Multiple WebFigures In an Array as the Output** .......... A-4

**Attaching a WebFigure** .......... A-4

**Referencing a WebFigure Attached to the Local Server** .......... A-4

**Referencing a WebFigure Attached to a Remote Server** .......... A-4

**Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up** .......... A-4

**Creating and Modifying a MATLAB Figure** .......... A-4

**Working with MATLAB Figures** .......... A-5

**Working with Images** .......... A-5

Building a Remotable Component Using the Deployment Tool . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-5

Building a Remotable Component Using the mcc Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-5

Using Native .NET Structure and Cell Arrays . . . . . . . . A-5

COM Components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-5

Utility Library Classes for COM Components . . . . . . . . A-6

Glossary

Index

# Getting Started


- "Product Overview" on page 1-2
- "Before You Use MATLAB® Builder NE" on page 1-3
- "Deploying a Component Using the Magic Square Example" on page 1-8
- "Magic Square Example: MATLAB Programmer Tasks" on page 1-10
- "Magic Square Example: .NET Developer Tasks" on page 1-20
- "Using the Magic Square Component in an Application" on page 1-36
- "Next Steps" on page 1-39

# Product Overview

## MATLAB Compiler Extension

MATLAB® Builder™ NE is an extension to MATLAB® Compiler™. The builder converts MATLAB® functions to .NET methods that encapsulate MATLAB code written by the MATLAB programmer. All MATLAB code to be compiled must take the form of a function. Each MATLAB Builder NE component contains one or more classes, each providing an interface to the MATLAB functions in the MATLAB code.

When you package and distribute the application to your users, you include supporting files generated by the builder as well as the MATLAB Compiler Runtime (MCR). For more information about the MCR, see "What Is The MATLAB Compiler Runtime (MCR)?" in the MATLAB Compiler documentation.

For more information about how this product works with MATLAB Compiler, see "Writing Deployable MATLAB Code".

# Before You Use MATLAB Builder NE

**In this section...**

"Your Role in the .NET Application Deployment Process" on page 1-3
"What You Need to Know" on page 1-4
"Install Products, Compilers, and IDEs" on page 1-5
"Install Microsoft .NET Framework" on page 1-5
"Consider Architecture of Deployment Targets" on page 1-6
"Evaluate Dependencies and Non-Compilable Code" on page 1-6
"For More Information" on page 1-6
"Troubleshooting Installation Problems" on page 1-7

## Your Role in the .NET Application Deployment Process

Depending on the size of your organization, you may play one or more roles in the process of successfully deploying a .NET application. For example, your role may be to:

- Analyze user requirements and satisfy them by writing a program in MATLAB code (MATLAB programmer)
- Implement the infrastructure needed to successfully deploy a .NET application to the Web (middle-tier developer)
- Create a remotable component that can be shared across distributed systems (.NET developer)
- Perform tasks associated with numerous roles, usually within a smaller organization (end-to-end developer)

The table Application Deployment Roles, Goals, and Tasks on page 1-4 describes some of the different roles, or jobs, that MATLAB Builder NE users typically perform and which tasks they would most likely perform when running the examples in this documentation.

**Application Deployment Roles, Goals, and Tasks**

| Role | Knowledge Base | Responsibilities | Task To Achieve Goal |
| --- | --- | --- | --- |
| **MATLAB programmer** | • MATLAB expert<br>• No IT experience<br>• No access to IT systems | • Develops models; implements in MATLAB<br>• Uses tools to create a component that is used by the .NET programmer | See "Magic Square Example: MATLAB Programmer Tasks" on page 1-10. |
| **.NET developer** | • Little or no MATLAB experience<br>• Moderate IT experience<br>• .NET expert<br>• Minimal access to IT systems | • Integrates deployed component with the rest of the .NET application<br>• Integrates deployed MATLAB Figures with the rest of the .NET application | See "Magic Square Example: MATLAB Programmer Tasks" on page 1-10. |

## What You Need to Know

To use the MATLAB Builder NE product, specific requirements exist for each user role.

| Role | Requirements |
|---|---|
| **MATLAB programmer** | • A basic knowledge of MATLAB, and how to work with:<br>- MATLAB data types<br>- MATLAB structures |
| **.NET developer** | • Exposure to:<br>- A CLS-compliant programming language<br>- .NET Framework<br>• Knowledge of object-oriented programming concepts |

## Install Products, Compilers, and IDEs

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

- MATLAB
- MATLAB Compiler
- MATLAB Builder NE
- An *Integrated Development Environment (IDE)* such as Microsoft® Visual Studio®

---

**Note** Log in with administrator privileges before installing these products.

---

For more information about product installation and requirements, see "Installation and Configuration" in the MATLAB Compiler documentation.

## Install Microsoft .NET Framework

Install the supported version of the Microsoft® .NET Framework. Your ability to use the latest builder functionality often depends on having the most current version of the framework installed.

See "Supported Microsoft .NET Framework Versions" on page 3-2 for details.

## Consider Architecture of Deployment Targets

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

You cannot deploy applications developed with one architecture to computers running a different architecture.

## Evaluate Dependencies and Non-Compilable Code

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 Chapter 2, "Writing Deployable MATLAB Code", and in particular "Guidelines for Writing Deployable MATLAB Code" on page 2-15 in this User's Guide.

## For More Information

| If you want to... | See... |
|---|---|
| Deploy a .NET component | "Deploying a Component Using the Magic Square Example" on page 1-8 |
| Deploy a COM component | "Magic Square Example" on page 15-2 for COM Builder |
| Deploy a component on the Web | Chapter 7, "Deploying a MATLAB Figure Over the Web Using WebFigures" |
| Deploy a figure or image | Chapter 8, "Working with MATLAB Figures and Images" |
| Deploy a *remotable component* | Chapter 9, "Sharing Components Across Distributed Applications Using .NET Remoting" |

## Troubleshooting Installation Problems

| For these issues... | Solution |
|---|---|
| `No compatible version of .NET framework found` | Install the supported Microsoft .NET Framework version. See "Supported Microsoft .NET Framework Versions" on page 3-2 |

# Deploying a Component Using the Magic Square Example

## About This Example

This example shows you how to transform a MATLAB function into a deployable MATLAB Builder NE component.

The Magic Square example shows you how to create a .NET component named `MagicSquareComp`, which contains the `MLTestClass` class and other files needed to deploy your application.

The `MLTestClass` wraps a MATLAB function, `makesquare`, which computes a magic square.

---

**Note** The examples here use the Windows® `deploytool` GUI, a graphical front-end interface to MATLAB Compiler software. For information about how to perform these tasks using the command-line interface to MATLAB Compiler software, see the `mcc` reference page in this User's Guide.

---

### What Is 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.

### How Do I Access the Examples?

For information on accessing the example files in the product, see "Copying the Example Files" on page 1-11.

### For More Information

| If you want to... | See... |
|---|---|
| See a complete code sample of integrating the Magic Square component into a .NET application | "Using the Magic Square Component in an Application" on page 1-36 |
| Deploy a Web component | Chapter 7, "Deploying a MATLAB Figure Over the Web Using WebFigures" |
| Deploy an existing figure or image | Chapter 8, "Working with MATLAB Figures and Images" |
| Deploy a *remotable component* | Chapter 9, "Sharing Components Across Distributed Applications Using .NET Remoting" |

# Magic Square Example: MATLAB Programmer Tasks

**In this section...**


"Copying the Example Files" on page 1-11

"Testing the MATLAB File You Want to Deploy" on page 1-11

"Building Your Component" on page 1-13

"Packaging Your Component (Optional)" on page 1-15

"Copying the Package You Created" on page 1-19


### MATLAB Programmer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| **MATLAB programmer** | • MATLAB expert<br>• No IT experience<br>• No access to IT systems | • Develops models; implements in MATLAB<br>• Uses tools to create a component that is used by the .NET developer |

The MATLAB programmer usually performs the following tasks.

### Key Tasks for the MATLAB Programmer

| Task | Reference |
|---|---|
| Prepare to run the example by copying the MATLAB example files into a work folder. | "Copying the Example Files" on page 1-11 |
| Test the MATLAB code to ensure it is suitable for deployment. | "Testing the MATLAB File You Want to Deploy" on page 1-11 |
| Create a .NET component (encapsulating your MATLAB code in a .NET class) by running the `deploytool`. | "Building Your Component" on page 1-13 |

**Key Tasks for the MATLAB Programmer (Continued)**

| Task | Reference |
|---|---|
| Optionally, use the Packaging Tool to wrap all deliverable files in one bundle. | "Packaging Your Component (Optional)" on page 1-15 |
| Copy the output so the .NET programmer can work with it further. | "Copying the Package You Created" on page 1-19 |

## Copying the Example Files

Prepare to run the example by copying needed files into your work area as follows:

1. Navigate to `matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\MagicSquareExample`.

   ---

   **Tip** `matlabroot` is the MATLAB root folder (where MATLAB is installed). To find the value of this variable on your system, type `matlabroot` at a MATLAB command prompt.

   ---

2. Copy the `MagicSquareExample` folder to a work area, if desired (for example, `D:\dotnetbuilder_examples`). Avoid using spaces in your folder names, if possible. The example files should now reside in `D:\dotnetbuilder_examples\MagicSquareExample`.

---

**Note** The MATLAB Builder NE examples reside in `matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET`. This example assumes the work folder is on drive `D:`.

---

## Testing the MATLAB File You Want to Deploy

In this example, you test a precreated MATLAB file (`magicsquare.m`) containing the predefined MATLAB function `magic`, in order to have a baseline to compare to the results of the function when it is finally wrappered as a deployable .NET component.

**1** Using MATLAB, locate the `magicsquare.m` file at `D:\dotnetbuilder_examples\MagicSquareExample\MagicSquareComp`. This file has the following contents:

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

%   Copyright 2001-2010 The MathWorks, Inc.

y = magic(x);
```

**2** At the MATLAB command prompt, enter `makesquare(5)`, and view the results. The output should appear 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
```

## For More Information

| If you want to... | See... |
|---|---|
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, “Writing Deployable MATLAB Code” |

## Building Your Component

You create a .NET component by using the Deployment Tool GUI to build a .NET class that wraps around the sample MATLAB code discussed in "Testing the MATLAB File You Want to Deploy" on page 1-11.

Use the following information when creating your component as you work through this example:

| | |
|---|---|
| Project Name | `makeSqr` |
| Class Name | `MLTestClass` |
| File to compile | `makesquare.m` |

1 Start MATLAB if you have not done so already.

2 Type `deploytool` at the command prompt, and press **Enter**. The `deploytool` GUI opens.

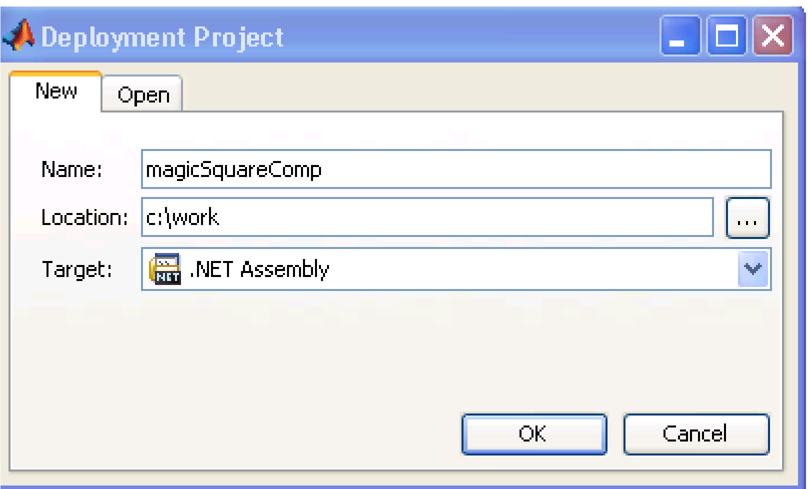
3 Create a deployment project using the Deployment Project dialog box:

   a Type the name of your project in the **Name** field.

   b Enter the location of the project in the **Location** field. Alternately, navigate to the location.

   c Select the target for the deployment project from the **Target** drop-down menu.

   d Click **OK**.




**Creating a .NET Project**

4 On the **Build** tab:

- If you are building a COM application, click **Add files**.
- If you are building a .NET application, click **Add class**. Type the name of the class in the Class Name field, designated by the letter c:

  For this class, add methods you want to compile (your MATLAB by clicking **Add files**. To add another class, click **Add class**.

- You may optionally add supporting files. For examples of these files, see the `deploytool` Help. To add these files, in the Shared Resources and Helper Files area:

  **e** Click **Add files/directories**

  **f** Click **Open** to select the file or files.

5 When you complete your changes, click the Build button ( ).

### For More Information

| If you want to... | See... |
|---|---|
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, "Writing Deployable MATLAB Code" |

## Packaging Your Component (Optional)

Bundling the .NET component with additional files you can distribute to users is called *packaging*. You perform this step using the packaging function of `deploytool`. If you are creating a shared component and want to include additional code with the component, you must perform this step.

The package process zips the following files into a single self-extracting executable, `componentName.exe`:

- *`componentName`*`.dll`
- *`componentName`*`Native.dll`

---

**Note** See Chapter 9, "Sharing Components Across Distributed Applications Using .NET Remoting" for more information on using the native .NET API.

---

- *`componentName`*`.xml`
- *`componentName`*`.pdb` (if the **Debug** option is selected)
- `MCRInstaller.exe` (if the **Include MCR** option is selected)

- `_install.bat` (script run by the self-extracting executable)

---

**Tip** Instead of performing the steps below, you can alternately copy the `distrib` folder and the MCR Installer to a local folder of your choice.

---

1 On the **Package** tab, add the MATLAB Compiler Runtime (MCR). To do so, click **Add MCR**, and choose one of the two options described in the following table.

| Option | What Does This Option Do? | When Should I Use This Option? |
|---|---|---|
| **Embed the MCR in the package** | This option physically copies the MCR Installer file into the package you create. | • You have a limited number of end users who deploy a small number of applications at sporadic intervals<br>• Your users have no intranet/network access<br>• Resources such as disk space, performance, and processing time are not significant concerns<br><br>**Note** Distributing the MCR Installer with each application requires more resources. |
| **Invoke the MCR from a network location** | This option lets you add a link to an MCR Installer residing on a local area network, allowing you to invoke the installer over the network, as opposed to copying the installer physically into the | • You have a large number of end users who deploy applications frequently<br>• Your users have intranet/network access<br>• Resources such as disk space, performance, |

| Option | What Does This Option Do? | When Should I Use This Option? |
|---|---|---|
| | deployable package.<br>This option sets up a script to install the MCR from a specified network location, saving time and resources when deploying applications. | and processing time are significant concerns for your organization<br><br>If you choose this option, modify the location of the MCR Installer, if needed. To do so, select the **Preferences** link in this dialog box, or change the **Compiler** option in your MATLAB Preferences.<br><br>**Caution** Before selecting this option, consult with your network or systems administrator. Your administrator may already have selected a network location from which to run the MCR Installer. |

For more information about the role the MCR plays in the deployment process, see "Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)" on page 4-3.

2. Next, add others files you feel may be useful to end users. To package additional files or folders, click **Add file/directories**, select the file or folder you want to package, and click **Open**.

3. In the Deployment Tool, click the Packaging button ().

4. For Windows, the package is a self-extracting executable. On platforms other than Windows, the package is delivered as a `.zip` file. Verify that the contents of the `distrib` folder contains the files you specified.

---

**Note** When the self-extracting executable is uncompressed on a system, `VCREDSIT_X86` is installed. `VCREDSIT_X86` installs run-time components of Microsoft Visual C++ libraries necessary for running Visual C++ applications.

---

### About the MCR and the MCR Installer

The *MATLAB Compiler Runtime (MCR)* is an execution engine made up of the same shared libraries MATLAB uses to enable the execution of MATLAB files on systems without an installed version of MATLAB. In order to deploy a component, you *package* the MCR along with it. Before you utilize the MCR on a system without MATLAB, run the *MCR installer*.

The installer does the following:

1. Installs the MCR (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 MCR

### How the MCR Is Shared Among Classes

The builder creates a single MCR instance for each Builder class in an application. All class instances share a single MATLAB workspace and share global variables in the MATLAB files used to build the component. This MCR instance is reused and shared among all subsequent class instances within the component. Such reuse and sharing results in more efficient memory usage and elimination of MCR startup cost in each subsequent class instantiation.

### For More Information

| If you want to... | See... |
|---|---|
| Build (compile) a component from your MATLAB code | "Building Your Component" on page 1-13 |
| Integrate your component into a production environment | "Integrating Your Component into a .NET Class Using Microsoft® Visual Studio" on page 1-25 |
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, "Writing Deployable MATLAB Code" |

## Copying the Package You Created

Copy the package that you created to the local folder of your choice, or send them directly to the .NET developer.

# Magic Square Example: .NET Developer Tasks

| In this section... |
| --- |
| "Gathering Files Needed for Deployment" on page 1-21<br>"Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)" on page 4-3<br>"Integrating Your Component into a .NET Class Using Microsoft® Visual Studio" on page 1-25<br>"Building and Testing the .NET Application with Microsoft® Visual Studio" on page 1-34 |

The following tasks are usually performed by the .NET developer.

**.NET Developer**

| Role | Knowledge Base | Responsibilities |
| --- | --- | --- |
| **.NET Developer** | • Little to no MATLAB experience<br>• Moderate IT experience<br>• .NET expert<br>• Minimal access to IT systems | • Integrates deployed component with the rest of the .NET application |

**Key Tasks for the .NET Developer**

| Task | Reference |
| --- | --- |
| Ensure you have the needed files from the MATLAB programmer. | "Gathering Files Needed for Deployment" on page 1-21 |
| Install your application on target computers without MATLAB (installing the MCR). | "Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)" on page 4-3 |

**Key Tasks for the .NET Developer (Continued)**

| Task | Reference |
|---|---|
| Integrate classes generated by the MATLAB Builder NE product into existing .NET applications. | "Integrating Your Component into a .NET Class Using Microsoft® Visual Studio" on page 1-25 |
| Verify your .NET application works as expected in your end user's deployment environment. | "Building and Testing the .NET Application with Microsoft® Visual Studio" on page 1-34 |

## Gathering Files Needed for Deployment

Before beginning, verify you have the files the MATLAB programmer packaged, listed in "Packaging Your Component (Optional)" on page 1-15.

The package is a self-extracting executable. Paste it in a folder on the development machine and run it. If you are using a `.zip` file bundled with WinZip, unzip and extract the contents to the development machine.

## Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)

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

### About the MCR and the MCR Installer

The *MATLAB Compiler Runtime (MCR)* is an execution engine made up of the same shared libraries MATLAB uses to enable the execution of MATLAB files on systems without an installed version of MATLAB. In order to deploy a component, you *package* the MCR along with it. Before you utilize the MCR on a system without MATLAB, run the *MCR installer*.

The installer does the following:

1 Installs the MCR (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 MCR

### Before You Install the MCR

1. Since installing the MCR requires write access to the system registry, ensure you have administrator privileges to run the MCR Installer.

2. The version of the MCR that runs your application on the target computer must be compatible with the version of MATLAB Compiler that built the component.

---

**Caution** If an MCR does not match the version of the instance of MATLAB that built the component, an inoperable application and unpredictable results can result.

---

### Including the MCR installer With Your Deployment Package

Include the MCR in your deployment by using the Deployment Tool.

On the **Package** tab of the `deploytool` interface, click **Add MCR**.

---

**Note** For more information about additional options for including the MCR Installer (embedding it in your package or locating the installer on a network share), see "Packaging Your Deployment Application (Optional)" in the *MATLAB Compiler User's Guide* or in your respective Builder User's Guide.

---

### Installing the MCR and Setting System Paths

To install the MCR, perform the following tasks on the target machines:

1. If you added the MCR during packaging, open the package to locate the installer (`MCRInstaller.exe`). Otherwise, run the command `mcrinstaller` to display the locations where you can download the installer.

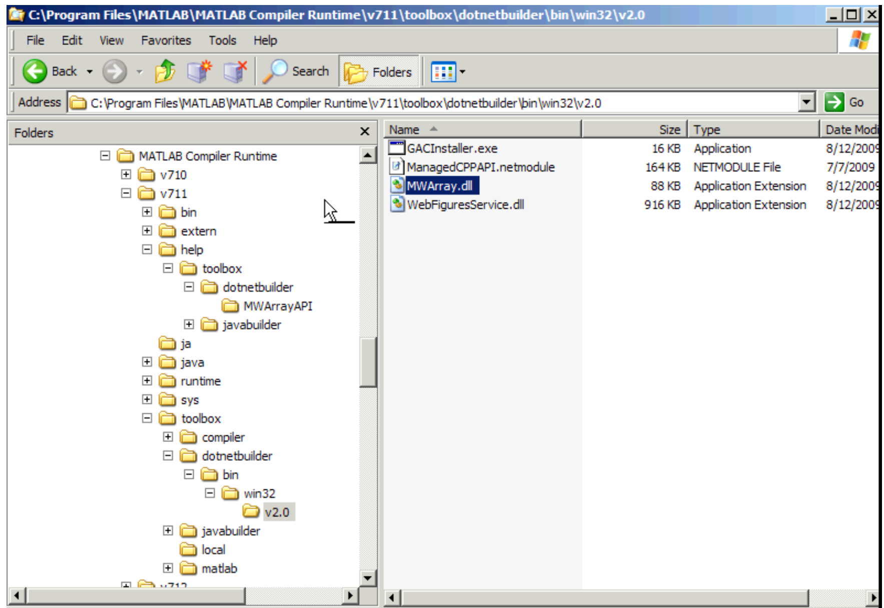
2. If you are running on a platform other than Windows, set the system paths on the target machine. Setting the paths enables your application to find the MCR.

   Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See "Using Run Script to Set MCR Paths" in the appendix "Using MATLAB Compiler on UNIX®" in the *MATLAB Compiler User's Guide* for more information.

**Where to find the MWArray API.** The MCR also includes `MWArray.dll`, which contains an API for exchanging data between your applications and the MCR. You can find documentation for this API in the `Help` folder of the installation.

On target machines where the MCR Installer is run, the MCR Installer puts the `MWArray` assembly in *`installation_folder`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`*.

See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.




**Sample Directory Structure of the MCR Including MWArray.dll**

## For More Information

| If you want to... | See... |
|---|---|
| Find more information about the MATLAB Compiler C++ API | "C++ Utility Library Reference" in this User's Guide |
| Find more information about the `MWArray` class library (the .NET API) | MATLAB Builder NE "Documentation Set" at the MathWorks Web site |

| If you want to... | See... |
|---|---|
| Find more information about the `MWArray` class library (the Java™ API) | MATLAB Builder JA"Documentation Set" at the MathWorks Web site |
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, "Writing Deployable MATLAB Code" |
| Learn more about the MATLAB Compiler Runtime (MCR) | "Working with the MCR" in the *MATLAB Compiler User's Guide* |

## Integrating Your Component into a .NET Class Using Microsoft Visual Studio


- "Creating a Microsoft® Visual Studio Project" on page 1-26
- "Creating a Reference to Your Component in C#/.NET Code" on page 1-26
- "Creating a Reference to the MWArray API" on page 1-27
- "Making .NET Namespaces Available for Your Generated Component and MWArray Libraries" on page 1-27
- "Initializing Your Classes" on page 1-29
- "Instantiating Your Classes" on page 1-30
- "Invoking the Component" on page 1-31
- "Handling Errors Using Try-Catch Blocks" on page 1-32

- "For More Information" on page 1-34

### Creating a Microsoft Visual Studio Project

To create a Microsoft Visual Studio project:

1. Open Microsoft Visual Studio
2. Click **File > New > Project**.
3. In the New Project dialog, select the project type and template you want to use.

   For example, if you want to create a C# Console Application, select **Windows** in the **Visual C#** branch of the **Project Type** pane, and select the **Console Application** template from the **Templates** pane.
4. Type the name of the project in the **Name** field (**MainApp**, for example).

### Creating a Reference to Your Component in C#/.NET Code

Create a reference in your code to the component that you just built.

To do so, in Microsoft Visual Studio, perform the following steps:

1. In the Solution Explorer pane within Microsoft Visual Studio (usually on the right side), select the name of your project, highlighting it.
2. Right-click and select **Add Reference**.
3. In the Add Reference dialog box, select the **Browse** tab. Locate the `distrib` folder using this dialog box. When you are finished the path to your `distrib` folder should be in the **File Name:** field. Click **OK** when done to open the `distrib` folder.
4. After you access the folder, select your component's executable file (in the case of the standalone example, a `.dll` file). Click **OK**.

### Creating a Reference to the MWArray API

You also need to reference `MWArray.dll`, which has already been registered with the *Global Assembly Cache (GAC)* (if you installed the MCR). The GAC is a machine-wide .NET assembly cache for Microsoft's CLR platform.

To create the reference, in Microsoft Visual Studio, perform the following steps:

1. In the Solution Explorer pane within Microsoft Visual Studio (usually on the right side), select the name of your project, and highlight it.

2. Right-click and select **Add Reference**.

3. In the Add Reference dialog box, select the **.NET** tab. Locate **MathWorks, .NET MWArray API**, and select it. Click **OK**.

---

**Tip** If you cannot locate the **MathWorks, .NET MWArray API**, it is likely the MCR was not installed correctly. See "Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)" on page 4-3 for more information.

---

### Making .NET Namespaces Available for Your Generated Component and MWArray Libraries

Make pertinent namespaces available to your application by adding the following `using` statements to your C#/.NET code:

```
using com.component_name;
using MathWorks.MATLAB.NET.Arrays;
using MathWorks.MATLAB.NET.Utility;
```

---

**Tip** When you are adding these statements to your code, you the following best-practices.

- Terminate each using statement with a semi-colon (;).
- Add references to the MWArray API. Microsoft Visual Studio's auto-completion feature, Intellisense™, can provide you completion tips as you write your code, but only if you first add references. See "Creating a Reference to Your Component in C#/.NET Code" on page 1-26 and "Creating a Reference to the MWArray API" on page 1-27 for more information.

---

**Specifying Component Assembly and Namespace.** To use the component assembly generated using the MATLAB Builder NE product from the client application, you must

- Reference the MATLAB data conversion assembly and specify the namespace in your application, as shown:

  ```
  using MathWorks.MATLAB.NET.Arrays;
  ```

- Reference the namespace for the builder assembly generated for your particular component and specify the namespace in your application, for example:

  ```
  using MyComponentName;
  ```

---

**Note** The builder supports nested namespaces.

---

Suppose you named the component you created `MyComponentName` and you want to use it in a program named `MyApp.cs`. Here are the statements to use at the beginning of `MyApp.cs`:

```
using System;
using MathWorks.MATLAB.NET.Arrays;
using MyComponentName;
```

## Initializing Your Classes

As a best practice, initialize your classes before you use them.

For example, you start by initializing an abstract class, `MLTestClass`, and then writing the code as follows.

```
using System;
using System.Collections.Generic;
using System.Text;

using com.demo;
using MathWorks.MATLAB.NET.Arrays;
using MathWorks.MATLAB.NET.Utility;


namespace MainApp
{
    class Program
    {
        static void Main(string[] args)
        {

            MLTestClass obj = null;
            MWNumericArray input = null;
            MWNumericArray output = null;
            MWArray[] result = null;




        }
    }
}
```

**Initializing Classes**

In this example, you perform the following initializations:

- Initializing classes to handle all input and output arguments (`MWNumericArray input = null` and `MWNumericArray output = null`). `MWNumericArray` is an interface to MATLAB's `numeric` type.
- Initializing a class to handle `MWArrays` that the program returns (`MWArray[] result = null` — an array of `MWArrays`).

### Instantiating Your Classes

After your classes have been initialized, you write code to create *objects* (instances of the class) using *instantiation* (creating an instance of a class). Instantiating a class requires use of the `new` keyword.

In this example, you instantiate `MLTestClass` with `obj = new MLTestClass();`:

MainApp - Microsoft Visual Studio

Program.cs*

```
using System;
using System.Collections.Generic;
using System.Text;

using com.demo;
using MathWorks.MATLAB.NET.Arrays;
using MathWorks.MATLAB.NET.Utility;


namespace MainApp
{
    class Program
    {
        static void Main(string[] args)
        {

            MLTestClass obj = null;
            MWNumericArray input = null;
            MWNumericArray output = null;
            MWArray[] result = null;


            obj = new MLTestClass();




        }
    }
}
```

**Instantiating a Class**

**Creating an Instance of a Class.** As with any .NET class, you need to create an instance of the classes you create with the MATLAB Builder NE product before you can use them in your program.

Suppose you build a component with a class named `MyComponentClass`. Here is an example of creating an instance of that class:

```
MyComponentClass classInstance = new MyComponentClass();
```

See “How the MCR Is Shared Among Classes” on page 1-18 for additional information about what happens when you instantiate classes.

## Invoking the Component

After you complete the tasks of initializing and instantiating the classes you are working with, invoke the `makeSqr` component you built with MATLAB Builder NE in “Building Your Component” on page 1-13.

Invoke the component method using a signature containing both the number of output arguments expected and the number of input arguments the MATLAB function requires. In this example, using `makeSqr`, the number of input arguments is 1 and the number of output arguments is 5:

```
obj.makeSqr(1, 5);
```

**Using Implicit Conversion.** Make use of implicit conversion, from .NET types to MATLAB types, by passing the native C# value directly to `makeSqr` using the `input` argument, as follows:

```
input = new MWNumericArray(5);
obj.makeSqr(1, input);
```

**Using Implicit Constructors.** You can exploit implicit constructors supplied with `MWArray` classes to save writing code.

For example, in this example, you can directly define your `input` as 5 (`input` being identified previously as MATLAB numeric type `MWNumericArray`).

```
namespace MainApp
{
    class Program
    {
        static void Main(string[] args)
        {

            MLTestClass obj = null;
            MWNumericArray input = null;
            MWNumericArray output = null;
            MWArray[] result = null;


            obj = new MLTestClass();

            input = 5;
            obj.makeSqr(1, input);
```

**Defining Input Using an Implicit Constructor**

**Handling Output from MWArray.** The `makeSqr` method returns an array of `MWArrays`

Extract the Magic Square you created from the first indice of `result` and print the output, as follows:

```
output = (MWNumericArray)result[0];
Console.WriteLine(output);
```

## Handling Errors Using Try-Catch Blocks

Because class instantiation and method invocation make their exceptions at run-time, you enclose your code in a try-catch block to handle errors.

```
namespace MainApp
{
    class Program
    {
        static void Main(string[] args)
        {

            MLTestClass obj = null;
            MWNumericArray input = null;
            MWNumericArray output = null;
            MWArray[] result = null;

            try
            {

                obj = new MLTestClass();

                input = 5;
                result = obj.makeSqr(1, input);

                output = (MWNumericArray)result[0];
                Console.WriteLine(output);
            }
            catch (Exception)
            {

                throw;
            }


        }
```

**Using a Try-Catch Block**

## For More Information

| If you want to... | See... |
|---|---|
| Perform advanced integration tasks | Chapter 4, "Integrating Your .NET Component" |
| Convert native data types to MATLAB data types | "Manually Converting Native Data Types to MATLAB Data Types" on page 4-11 |
| Automatic casting to MATLAB data types | "Automatic Casting to MATLAB Types" on page 4-10 |

| If you want to... | See... |
|---|---|
| Learn more about building your component | Chapter 3, "Building Your .NET Component" |
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, "Writing Deployable MATLAB Code" |

## Building and Testing the .NET Application with Microsoft Visual Studio

After you finish writing your code, you build and run it with Microsoft Visual Studio:

1. To build the application, select **Build > Build Solution**.

2. To run the application, select **Debug > Start Without Debugging**.

### For More Information

| If you want to... | See... |
|---|---|
| Perform advanced integration tasks | Chapter 4, "Integrating Your .NET Component" |
| Convert native data types to MATLAB data types | "Manually Converting Native Data Types to MATLAB Data Types" on page 4-11 |
| Automatic casting to MATLAB data types | "Automatic Casting to MATLAB Types" on page 4-10 |

| If you want to... | See... |
|---|---|
| Learn more about building your component | Chapter 3, "Building Your .NET Component" |
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, "Writing Deployable MATLAB Code" |

# Using the Magic Square Component in an Application

1 Write source code for an application that uses the .NET component created in “Building Your Component” on page 1-13.

The C# source code for the sample application for this example is in `MagicSquareExample\MagicSquareCSApp\MagicSquareApp.cs`.

The program listing is shown here.

---

**Tip** Although MATLAB Builder NE generates C# code for the `MagicSquare` component and the sample application is in C#, applications that use the component do not need to be coded in C#. You can access the component from any CLS-compliant .NET language. For examples, see Chapter 6, “Sample Applications (Microsoft® Visual Basic .NET)”.

---

### MagicSquareApp.cs

```
// *******************************************************************************
//
// MagicDemoApp.cs
//
// This file is an example application for the MATLAB Builder NE product.
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *******************************************************************************

using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using MagicDemoComp;


namespace MathWorks.Demo.MagicSquareApp
{
```

```
/// <summary>
/// The MagicSquareApp demo class computes a magic square of the user-specified size.
/// </summary>
/// <remarks>
/// args[0] - a positive integer representing the array size.
/// </remarks>
class MagicDemoApp
  {
     #region MAIN

   /// <summary>
   /// The main entry point for the application.
   /// </summary>
   [STAThread]
   static void Main(string[] args)
     {
         MWNumericArray arraySize= null;
         MWNumericArray magicSquare= null;

         try
           {
             // Get user-specified command line arguments or set default
              arraySize= (0 != args.Length) ? System.Double.Parse(args[0]) : 4;

             // Create the magic square object
             MagicSquareClass magic= new MagicSquare();

             // Compute the magic square and print the result
             magicSquare= (MWNumericArray)magic.makesquare((MWArray)arraySize);

             Console.WriteLine("Magic square of order {0}\n\n{1}",
                                 arraySize, magicSquare);

             // Convert the magic square array to a two-dimensional native double array
             double[,] nativeArray= (double[,])magicSquare.ToArray(MWArrayComponent.Real);

             Console.WriteLine("\nMagic square as native array:\n");

             // Display the array elements:
             for (int i= 0; i < (int)arraySize; i++)
```

```
                for (int j= 0; j < (int)arraySize; j++)
                  Console.WriteLine("Element({0},{1})= {2}", i, j, nativeArray[i,j]);

              Console.ReadLine();  // Wait for user to exit application
            }

          catch(Exception exception)
            {
              Console.WriteLine("Error: {0}", exception);
            }
        }
#endregion
     }
 }
```

**2** Build the application using Visual Studio® .NET.

---

**Note** In the project file for this example, the `MWArray` assembly and the magic square component assembly have been prereferenced. Any references preceded by an exclamation point require you to remove the reference and rereference the affected assembly.

---

---

**Note** Microsoft .NET Framework version 2.0 is not supported by Visual Studio 2003.

---

**a** Open the project file for the Magic Square example (`MagicSquareCSApp.csproj`) in Visual Studio .NET.

**b** If necessary, add a reference to the `MWArray` component in *`matlabroot`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`*.

See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

**c** If necessary, add a reference to the Magic Square component (`MagicSquareComp`), which is in the `distrib` subfolder.

# Next Steps

After you create and distribute the initial application, you probably want to continue to enhance that application. See the following documentation references for details about some of the more common tasks to perform as you develop your application.

| | |
|---|---|
| Writing .NET applications that can access .NET methods that encapsulate MATLAB code | Chapter 4, "Integrating Your .NET Component" |
| Sample applications that access methods developed in MATLAB | Chapter 5, "Sample Applications (C#)" and Chapter 6, "Sample Applications (Microsoft® Visual Basic .NET)" |
| Deploying .NET components over the Web | Chapter 7, "Deploying a MATLAB Figure Over the Web Using WebFigures" |
| Creating a remotable component or learning about remotable components | Chapter 9, "Sharing Components Across Distributed Applications Using .NET Remoting" |

2

# Writing Deployable MATLAB Code


- “The MATLAB Application Deployment Products” on page 2-2
- “How the Deployment Products Process MATLAB Function Signatures” on page 2-4
- “The Application Deployment Products and the Deployment Tool” on page 2-8
- “Guidelines for Writing Deployable MATLAB Code” on page 2-15
- “Working with MATLAB Data Files Using Load and Save” on page 2-19

# The MATLAB Application Deployment Products

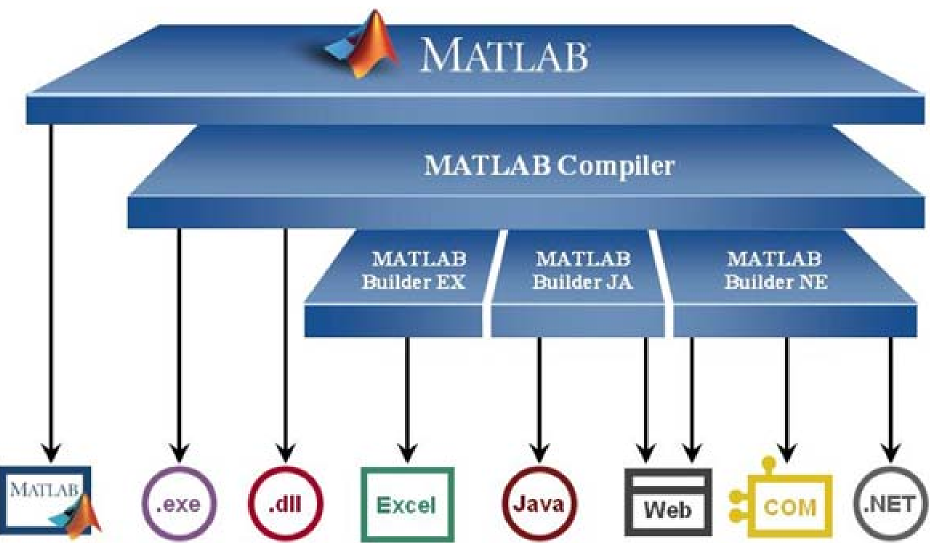
### MATLAB Programmer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| **MATLAB programmer** | • MATLAB expert<br>• No IT experience<br>• No access to IT systems | • Develops models; implements in MATLAB<br>• Uses tools to create a component that is used by the .NET developer |

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

### The MATLAB Suite of Application Deployment Products

| Product | Target | Stand-alones? | Function Libraries? | Graphical Apps? | Web Apps? | WebFigures? |
|---|---|---|---|---|---|---|
| MATLAB Compiler | C and C++ standalones | Yes | Yes | Yes | No | No |
| MATLAB Builder NE | C# .NET components Visual Basic COM components | No | Yes | Yes | Yes | Yes |
| MATLAB Builder JA | Java components | No | Yes | Yes | Yes | Yes |
| MATLAB Builder EX | Microsoft® Excel® add-ins | No | Yes | Yes | No | No |




**The MATLAB® Application Deployment Products**

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

# How the Deployment Products Process MATLAB Function Signatures

**In this section...**

"The MATLAB Function Signature" on page 2-4
"MATLAB Programming Basics" on page 2-5
"Returning MATLAB Data Types" on page 2-5

## The 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
```

## Returning MATLAB Data Types

MATLAB allows many different deployable data types. This section contains examples of how to work with figures. For an in-depth explanation of how to work with MATLAB primitive data types, see the *MATLAB External Interfaces* documentation.

## MATLAB Figures

Often, you are dealing with images displayed in a figure window, and not just string and numerical data. Deployed Web applications can support figure window data in a number of ways. By using the WebFigures infrastructure (see "Deploying a Java Component Over the Web" in the MATLAB Builder JA User's Guide or Chapter 7, "Deploying a MATLAB Figure Over the Web Using WebFigures" in the MATLAB Builder NE User's Guide), the respective builder marshalls the data for you.

Alternatively, you can take a snapshot of what is in the figure window at a given point and convert that data into the raw image data for a specific image type. This is particularly useful for streaming the images across the web.

### Returning Data from a WebFigure Window

WebFigures is a feature that enables you to embed dynamic MATLAB figures onto a Web page through a Builder JA or Builder NE component. This concept can be used with any data in a figure window.

As in the following example, you close the figure before the code is exited so that the figure does not "pop up," or appear later, in the deployed application. You do not need to specify any reorientation data when using WebFigures. If the figure is attached to the rest of the infrastructure, it will automatically pass, resize, and reorient accordingly.

```
%returns a WebFigure reference containing the
%data from the figure window
function resultWebFigure = getWebFigure
   f = figure;
   set(f,'Color',[.8,.9,1]);
   f = figure('Visible','off');
   surf(peaks);
   resultWebFigure = webfigure(f);
   close(f);
end
```

### Returning a Figure as Data

This approach is typically used for instances where WebFigures can't be used, or in a stateless application.

```
%We set the figure not to be visible since we are
%streaming the data out
%Notice how you can specify the format of the bytes,
%  .net uses unsigned bytes (uint8)
%  java uses signed bytes (int 8)
%This function allows you to specify the image format
%such as png, or jpg
function imageByteData = getSurfPeaksImageData(imageFormat)
  f = figure;
  surf(peaks);
  set(f, 'Visible', 'off');
  imageByteData = figToImStream(f, imageFormat, 'uint8');
  close(f);
end
```

### Reorienting a Figure and Returning It as Data

Sometimes you want the function to change the perspective on an image before returning it. This can be accomplished like this:

```
%We set the figure not to be visible since we are
%streaming the data out
%Notice how you can specify the format of the bytes,
%  .net uses unsigned bytes (uint8)
%  java uses signed bytes (int 8)
%This function allows you to specify the image format
%such as png, or jpg
function imageData =
     getImageDataOrientation(width, height, rotation,
                                   elevation, imageFormat)
  f = figure('Position', [0, 0, width, height]);
  surf(peaks);
  view([rotation, elevation]);
  set(f, 'Visible', 'off');
  imageData = figToImStream (f, imageFormat, 'uint8');
  close(f);
end
```

# The Application Deployment Products and the Deployment Tool

**In this section...**

"What Is the Difference Between the Deployment Tool and the mcc Command Line?" on page 2-8

"How Does MATLAB® Compiler Software Build My Application?" on page 2-9

"What You Should Know About the Dependency Analysis Function (depfun)" on page 2-10

"Compiling MEX-Files, DLLs, or Shared Libraries" on page 2-10

"The Role of the Component Technology File (CTF Archive)" on page 2-11

## What Is the Difference Between the Deployment Tool and the mcc Command Line?

When you use the Deployment Tool (`deploytool`) GUI, you perform any function you would invoke using the MATLAB Compiler `mcc` command-line interface. The Deployment Tool 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`.

Deployment Tool advantages include:

- You perform related deployment tasks with a single intuitive GUI.
- You maintain related information in a convenient project file.
- Your project state persists between sessions.
- Your previous project loads automatically when the Deployment Tool starts.
- 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 the Dependency Analysis Function (`depfun`). 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.
   - File deployability — Whether the file is deployable outside of MATLAB

   For more information about `depfun`, see “What You Should Know About the Dependency Analysis Function (depfun)” on page 2-10.

3. Validates MEX-files. In particular, `mexFunction` entry points are verified. For more details about MEX-file processing, see “Compiling MEX-Files, DLLs, or Shared Libraries” on page 2-10.

4. Creates a CTF archive from the input files and their dependencies. For more details about CTF archives see “The Role of the Component Technology File (CTF Archive)” on page 2-11.

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. Invokes a third-party target-specific compiler to create the appropriate binary software component (a standalone executable, a Java JAR file, and so on).

For details about how MATLAB Compiler software builds your deployable component, see “How Does MATLAB® Compiler Software Build My Application?” on page 2-9.

## What You Should Know About the Dependency Analysis Function (depfun)

MATLAB Compiler uses a dependency analysis function (`depfun`) to determine the list of necessary files to include in the CTF package. Sometimes, this process generates a large list of files, particularly when MATLAB object classes exist in the compilation and `depfun` cannot resolve overloaded methods at compile time. Dependency analysis also processes `include`/`exclude` files on each pass (see the `mcc` flag "-a Add to Archive").

---

**Tip** To improve compile time performance and lessen application size, prune the path with "-N Clear Path", "-p Add Directory to Path". You can also specify **Toolboxes on Path** in the `deploytool` **Settings**

For more information about `depfun` and `addpath` and `rmpath`, see "Dependency Analysis Function (depfun) and User Interaction with the Compilation Path".

---

`depfun` searches for executable content such as:

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

`depfun` does not search for data files of any kind (except MAT files). You must manually `include` data files in the search

## Compiling MEX-Files, DLLs, or Shared Libraries

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

- Because `depfun` 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 options on the **Advanced** tab in the Deployment Tool under **Settings**.

- If you have any doubts that `depfun` 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 by using the options on the **Advanced** tab in the Deployment Tool under **Settings**.
- 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.

## The Role of the Component Technology File (CTF Archive)

Each application or shared library you produce using MATLAB Compiler has an associated Component Technology File (CTF) archive. The archive contains all the MATLAB based content (MATLAB files, MEX-files, and so on) associated with the component.

MATLAB Compiler also embeds a CTF archive in each generated binary. The CTF houses all deployable files. All MATLAB files encrypt in the CTF archive using the Advanced Encryption Standard (AES) cryptosystem.

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

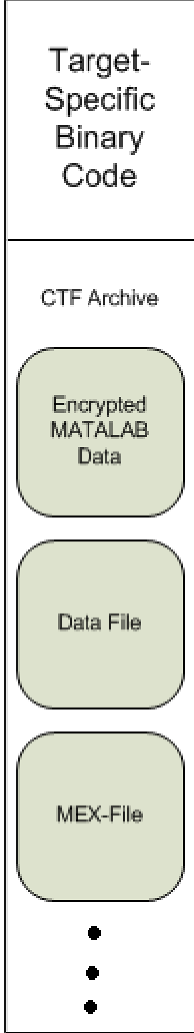
### Information on CTF Archive Embedding/Extraction and Component Cache

| Product | Refer to |
|---|---|
| MATLAB Compiler | “Overriding Default CTF Archive Embedding Using the MCR Component Cache” |
| MATLAB Builder NE | “Extracting the CTF Archive Manually Using the MCR Component Cache” on page 4-32 |

**Information on CTF Archive Embedding/Extraction and Component Cache (Continued)**

| Product | Refer to |
|---|---|
| MATLAB Builder JA | “Using MCR Component Cache and MWComponentOptions” |
| MATLAB Builder EX | “Overriding Default CTF Archive Embedding for Components Using the MCR Component Cache” |

Generated Component (EXE, DLL, SO, etc)



### Additional Details

Multiple CTF archives, such as those generated with COM, .NET, or Excel® components, 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 CTF archives into another CTF archive and distribute them.

All the MATLAB files from a given CTF archive associate with a unique cryptographic key. MATLAB files with different keys, placed in the same CTF 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 CTF archive.

MATLAB Compiler deleted the CTF 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 CTF archives. If you do, you can possibly strip the CTF archive from the binary, resulting in run-time errors for the driver application.

---

# Guidelines for Writing Deployable MATLAB Code

**In this section...**


"Compiled Applications Do Not Process MATLAB Files at Runtime" on page 2-15

"Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files" on page 2-16

"Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths" on page 2-16

"Gradually Refactor Applications That Depend on Noncompilable Functions" on page 2-17

"Do Not Create or Use Nonconstant Static State Variables" on page 2-17


## Compiled Applications Do Not Process MATLAB Files at Runtime

The MATLAB Compiler was designed so that you can deploy locked down functionality. 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, they both must be compiled in.

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

Some MATLAB toolboxes, such as the Neural Network Toolbox™ product, generate MATLAB code dynamically. Because the MCR 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 runtime 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 the next section 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.

For an example of using `isdeployed`, see "Passing Arguments to and from a Standalone Application".

## 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 MCR 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 with the builder components, you should be aware that an instance of the MCR 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 MCR 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 MCRs 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.

---

# Working with MATLAB Data Files Using Load and Save

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 CTF archive.

For more information about CTF archives, see "The Role of the Component Technology File (CTF Archive)" on page 2-11.

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 *`install_root`*`\extern\examples\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 -mC 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 ctfroot; All other folders will have the
%    folder
% structure included in the ctf 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 ctf and extracted under ctfroot. For example:
%   Data file
%     "c:\$matlabroot\examples\externdata\extern_data.mat"
%     will be added into ctf and extracted to
%  "$ctfroot\$matlabroot\examples\externdata\extern_data.mat".
%
% All mat/data files are unchanged after mcc runs. There is
% no excryption on these user included data files. They are
% included in the ctf 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 ctf;
    % 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');
```

3

# Building Your .NET Component


- "Supported Compilation Targets" on page 3-2
- "Using the Deployment Tool GUI to Build .NET Components" on page 3-4
- "Using the Command Line to Build .NET Components" on page 3-5
- "For More Information" on page 4-41

# Supported Compilation Targets

**In this section...**

".NET Component" on page 3-2

"COM Components" on page 3-3

**MATLAB Programmer**

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| **MATLAB programmer** | • MATLAB expert<br>• No IT experience<br>• No access to IT systems | • Develops models; implements in MATLAB<br>• Uses tools to create a component that is used by the .NET developer |

## .NET Component

MATLAB Builder NE supports compilation ("building") of .NET components through CLS compliant language wrapper generation.

### Common Language Specification (CLS) Compliancy

CLS is an acronym for *Common Language Specification*, a subset of language features supported by the *.NET common language runtime (CLR)*. MATLAB Builder NE classes are CLS compliant—they are designed to interoperate with all .NET programming languages.

Use the builder to package MATLAB functions so that .NET programmers can access them from any CLS-compliant language.

### Supported Microsoft .NET Framework Versions

As of this release, MATLAB Builder NE supports versions 2.0, 3.0, and 3.5 of the Microsoft .NET Framework.

All of these supported releases are based on the CLS 2.0 specification.

Specifying `0.0` as the .NET Framework indicates the latest version of the Framework installed on your system.

### Limitations of Support

MATLAB Builder NE provides a wide variety of support for various CLS-compliant types and objects, with the exception of MATLAB (MCOS) objects.

## COM Components

You can also use the builder to create Component Object Model, or COM, components. These components use a software architecture developed by Microsoft® to build component-based applications. COM objects expose interfaces that allow applications and other components to access the features of the objects. You access COM objects through Microsoft® Visual Basic®, C++, or any language that supports COM objects. For more information about creating and accessing COM components, see Chapter 13, "Creating and Installing COM Components" and Chapter 14, "Programming with COM Components Created by the MATLAB® Builder NE Product".

# Using the Deployment Tool GUI to Build .NET Components

For a complete example of how to build .NET components using the graphical interface, read "Deploying a Component Using the Magic Square Example" on page 1-8 in the Chapter 1, "Getting Started" chapter of this User's Guide.

For information about how the graphical interface differs from the command line interface, read "What Is the Difference Between the Deployment Tool and the mcc Command Line?" on page 2-8 in the chapter Chapter 2, "Writing Deployable MATLAB Code", also in this guide.

# Using the Command Line to Build .NET Components

**In this section...**

"Command-Line Syntax Description" on page 3-5

"Using the Command Line to Start the Deployment Tool GUI" on page 3-7

## Command-Line Syntax Description

Instead of using the Deployment Tool to create .NET components, you can use the mcc command.

The following command defines the complete mcc command syntax with all required and optional arguments used to create a .NET component. Brackets indicate optional parts of the syntax.

```
mcc  -W 'dotnet:component_name,class_name,
0.0|framework_version, Private|Encryption_Key_Path,local|remote'
file1 [file2...fileN][class{class_name:file1
[,file2,...,fileN]},... [-d output_dir_path] -T link:lib
```

---

**Note** For complete information about the mcc command, including the -W option, see mcc in the function reference section of this User's Guide. To learn more about the mcc command and all of its options, see the MATLAB Compiler documentation.

---

### Using the .NET Bundle Files to Simplify the Command

You can simplify the command line used to create .NET components. To do so, use the .NET Builder bundle file, named dotnet. Using this bundle file still requires that you pass in the five parts (including local|remote) of the -W argument text string; however, you do not have to specify the -T option.

The following example creates a .NET component called mycomponent containing a single .NET class named myclass with methods foo and bar. When used with the -B option, the word *dotnet* specifies the name of the predefined .NET Builder bundle file.

```
mcc -B 'dotnet:mycomponent,myclass,2.0,
    encryption_keyfile_path,local'
    foo.m bar.m
```

In this example, the builder uses the .NET Framework version 2.0 to compile the component into a shared assembly using the key file specified in `encryption_keyfile_path` to sign the shared component.

See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

### Example: Creating a .NET Component Namespace

The following example creates a .NET component from two MATLAB files `foo.m` and `bar.m`.

```
mcc  -B 'dotnet:mycompany.mygroup.mycomponent,myclass,0.0,Private,local'
 foo.m bar.m
```

The example creates a .NET component named `mycomponent` that has the following namespace: `mycompany.mygroup`. The component contains a single .NET class, `myclass`, which contains methods `foo` and `bar`.

To use `myclass`, place the following statement in your code:

```
using mycompany.mygroup;
```

See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

### Example: Adding Multiple Classes to a Component

The following example creates a .NET component that includes more than one class. This example uses the optional `class{...}` argument to the `mcc` command.

```
mcc  -B 'dotnet:mycompany.mycomponent,myclass,2.0,Private,local' foo.m bar.m
class{myclass2:foo2.m,bar2.m}
```

The example creates a .NET component named `mycomponent` with two classes:

- `myclass` has methods `foo` and `bar`
- `myclass2` has methods `foo2` and `bar2`

See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

## Using the Command Line to Start the Deployment Tool GUI

| Desired Results | Command |
| --- | --- |
| Start Deployment Tool GUI with the **New/Open** dialog box active | `deploytool` (default)<br>or<br>`deploytool -n` |
| Start Deployment Tool GUI and load *`project_name`* | `deploytool` *`project_name`*`.prj` |
| Start Deployment Tool command line interface and build `project_name` after initializing | `deploytool -win32 -build` *`project_name`*`.prj` |
| Start Deployment Tool command line interface and package `project_name` after initializing | `deploytool -package` *`project_name`*`.prj` |
| Display MATLAB Help for the `deploytool` command | `deploytool -?` |

# For More Information

| If you want to... | See... |
|---|---|
| Learn how to build a component and perform basic integration tasks using C# code | Chapter 1, "Getting Started" |
| • Basic MATLAB Programmer tasks<br>• How the deployment products process your MATLAB functions<br>• How the deployment products work together | Chapter 2, "Writing Deployable MATLAB Code" |
| Advanced integration tasks for the .NET Developer | Chapter 4, "Integrating Your .NET Component" |
| The MATLAB Component Runtime (MCR) | "Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)" on page 4-3 |

4

# Integrating Your .NET Component


- "Overview of Basic Integration Tasks" on page 4-2
- "Coding Your .NET Application" on page 4-7
- "Deploying Your .NET Component to End Users" on page 4-30
- "Accessing Your Component On Another Computer" on page 4-40
- "For More Information" on page 4-41

# Overview of Basic Integration Tasks

### .NET Developer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| **.NET Developer** | • Little to no MATLAB experience<br>• Moderate IT experience<br>• .NET expert<br>• Minimal access to IT systems | • Integrates deployed component with the rest of the .NET application |

In "Deploying a Component Using the Magic Square Example" on page 1-8, and in "Integrating Your Component into a .NET Class Using Microsoft® Visual Studio" on page 1-25 in particular, steps are illustrated that cover the basics of customizing your code in preparation for integrating your deployed .NET component into a large-scale enterprise application. These steps include:

- Installing the MATLAB Compiler Runtime (MCR) on end user computers
- Creating a Microsoft Visual Studio project
- Creating references to the component and to the `MWArray` API
- Specifying component assemblies and namespaces
- Initializing and instantiating your classes
- Invoking the component using some implicit data conversion techniques
- Handling errors using a basic try-catch block.

Watch a video that interactively demonstrates how to perform these basic steps on the MATLAB Builder NE product page.

## Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)

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

### About the MCR and the MCR Installer

The *MATLAB Compiler Runtime (MCR)* is an execution engine made up of the same shared libraries MATLAB uses to enable the execution of MATLAB files on systems without an installed version of MATLAB. In order to deploy a component, you *package* the MCR along with it. Before you utilize the MCR on a system without MATLAB, run the *MCR installer*.

The installer does the following:

1. Installs the MCR (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 MCR

### Before You Install the MCR

1. Since installing the MCR requires write access to the system registry, ensure you have administrator privileges to run the MCR Installer.

2. The version of the MCR that runs your application on the target computer must be compatible with the version of MATLAB Compiler that built the component.

---

**Caution** If an MCR does not match the version of the instance of MATLAB that built the component, an inoperable application and unpredictable results can result.

---

### Including the MCR installer With Your Deployment Package

Include the MCR in your deployment by using the Deployment Tool.

On the **Package** tab of the `deploytool` interface, click **Add MCR**.

---

**Note** For more information about additional options for including the MCR Installer (embedding it in your package or locating the installer on a network share), see "Packaging Your Deployment Application (Optional)" in the *MATLAB Compiler User's Guide* or in your respective Builder User's Guide.

---

### Installing the MCR and Setting System Paths

To install the MCR, perform the following tasks on the target machines:

1. If you added the MCR during packaging, open the package to locate the installer (`MCRInstaller.exe`). Otherwise, run the command `mcrinstaller` to display the locations where you can download the installer.

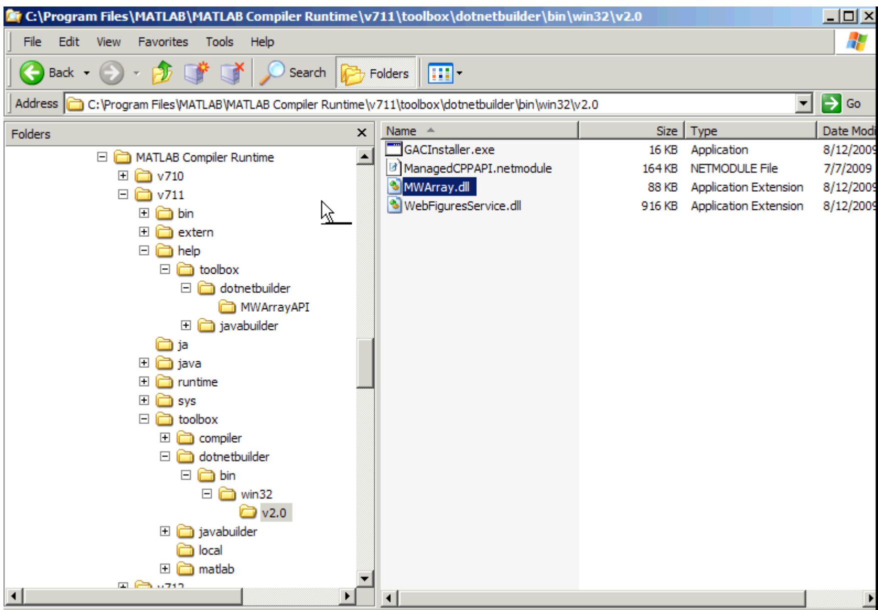
2. If you are running on a platform other than Windows, set the system paths on the target machine. Setting the paths enables your application to find the MCR.

   Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See "Using Run Script to Set MCR Paths" in the appendix "Using MATLAB Compiler on UNIX" in the *MATLAB Compiler User's Guide* for more information.

**Where to find the MWArray API.** The MCR also includes `MWArray.dll`, which contains an API for exchanging data between your applications and the MCR. You can find documentation for this API in the `Help` folder of the installation.

On target machines where the MCR Installer is run, the MCR Installer puts the `MWArray` assembly in *`installation_folder`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`*.

See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.




**Sample Directory Structure of the MCR Including MWArray.dll**

## For More Information

| If you want to... | See... |
|---|---|
| Find more information about the MATLAB Compiler C++ API | "C++ Utility Library Reference" in this User's Guide |
| Find more information about the `MWArray` class library (the .NET API) | MATLAB Builder NE "Documentation Set" at the MathWorks Web site |

| If you want to... | See... |
|---|---|
| Find more information about the `MWArray` class library (the Java API) | MATLAB Builder JA“Documentation Set” at the MathWorks Web site |
| • Perform basic MATLAB Programmer tasks<br>• Understand how the deployment products process your MATLAB functions<br>• Understand how the deployment products work together<br>• Explore guidelines about writing deployable MATLAB code | Chapter 2, “Writing Deployable MATLAB Code” |
| Learn more about the MATLAB Compiler Runtime (MCR) | “Working with the MCR” in the *MATLAB Compiler User’s Guide* |

# Coding Your .NET Application

**In this section...**


"About Your .NET Application and C# Code" on page 4-7

"Perform Data Conversion Where Required" on page 4-9

"Using MATLAB API Functions in a C# Program" on page 4-20

"Accessing Real or Imaginary Components Within Complex Arrays" on page 4-22

"Adding Fields to Data Structures and Data Structure Arrays" on page 4-24

"Using MATLAB Array Indexing" on page 4-24

"Blocking Execution of a Console Application that Creates Figures" on page 4-25

"Handling Errors" on page 4-27

"Using Dispose to Explicitly Free Resources" on page 4-28


## About Your .NET Application and C# Code

Before you begin integrating your component code with your .NET application, it is helpful to understand how the elements of the Deployment Tool project map to the class names in your generated wrapper code, and the naming conventions used for class and methods names in this code.

### Classes and Methods

The builder project contains the files and settings needed by the MATLAB Builder NE product to create a deployable .NET component. A project specifies information about classes and methods, including the MATLAB functions to be included.

The builder transforms MATLAB functions that are specified in the component's project to methods belonging to a *managed class*.

When creating a component, you must provide one or more class names as well as a component name. The component name also specifies the name of

the assembly that implements the component. The class name denotes the name of the class that encapsulates MATLAB functions.

To access the features and operations provided by the MATLAB functions, instantiate the managed class generated by the builder, and then call the methods that encapsulate the MATLAB functions.

### Component and Class Naming Conventions

Typically you should specify names for components and classes that will be clear to programmers who use your components. For example, if you are encapsulating many MATLAB functions, it helps to determine a scheme of function categories and to create a separate class for each category. Also, the name of each class should be descriptive of what the class does.

The *.NET Framework General Reference* recommends the use of *Pascal case* for capitalizing the names of identifiers of three or more characters. That is, the first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. For example:

```
MakeSquare
```

In contrast, MATLAB programmers typically use all lowercase for names of functions. For example:

```
makesquare
```

By convention, the MATLAB Builder NE examples use Pascal case.

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

### About Version Control

The builder supports the standard versioning capabilities provided by the .NET Framework.

---

**Note** You can make side-by-side invocations of multiple versions of a component within the same application only if they access the same version of the MCR.

---

## Perform Data Conversion Where Required

There are many instances when you may need to convert various native data types to types compatible with MATLAB. Use this section as a guideline to performing some of these basic tasks.

### Managing Data Conversion Issues with MATLAB Builder NE Data Conversion Classes

To support data conversion between managed types and MATLAB types, the builder provides a set of data conversion classes derived from the abstract class, `MWArray`.

The `MWArray` data conversion classes allow you to pass most native .NET value types as parameters directly without using explicit data conversion. There is an implicit cast operator for most native numeric and string types that will convert the native type to the appropriate MATLAB array.

When you invoke a method on a component, the input and output parameters are a derived type of `MWArray`. To pass parameters, you can either instantiate one of the `MWArray` subclasses explicitly, or, in many cases, pass the parameters as a *managed data type* and rely on the implicit data conversion feature of .NET Builder.

**Overview of Classes and Methods in the Data Conversion Class Hierarchy.** To support MATLAB data types, the MATLAB Builder NE product provides the `MWArray` data conversion classes in the MATLAB Builder NE `MWArray` assembly. You reference this assembly in your managed application to convert native arrays to MATLAB arrays and vice versa.

See the MWArray API documentation on the MATLAB Builder NE Documentaion Roadmap page (on the Web on in the product help) for full details on the classes and methods provided.

The data conversion classes are built as a class hierarchy that represents the major MATLAB array types.

---

**Note** See "Overview" on page 11-7 for an introduction to the classes and see MWArray Class Library Reference (available online only) for details about this class library.

---

The root of the hierarchy is the `MWArray` abstract class. The `MWArray` class has the following subclasses representing the major MATLAB types: `MWNumericArray`, `MWLogicalArray`, `MWCharArray`, `MWCellArray`, and `MWStructArray`.

`MWArray` and its derived classes provide the following functionality:

- Constructors and destructors to instantiate and dispose of MATLAB arrays
- Properties to get and set the array data
- Indexers to support a subset of MATLAB array indexing
- Implicit and explicit data conversion operators
- General methods

**Using Cell and Struct Arrays with MWArray.** You must use .NET Remoting to integrate .NET cell and struct arrays with MWArray.

See "The Native .NET Cell and Struct Example" on page 9-27 for more information and a complete end-to-end example.

## Automatic Casting to MATLAB Types

---

**Note** Because the conversion process is automatic (in most cases), you do not need to understand the conversion process to pass and return arguments with MATLAB Builder NE components.

---

In most instances, if a native .NET primitive or array is used as an input parameter in a C# program, the builder transparently converts it to an

instance of the appropriate `MWArray` class before it is passed on to the component method. The builder can convert most CLS-compliant string, numeric type, or multidimensional array of these types to an appropriate `MWArray` type.

---

**Note** This conversion is transparent in C# applications, but might require an explicit casting operator in other languages, for example, `op_implicit` in Visual Basic®.

---

Here is an example. Consider the .NET statement:

```
result = theFourier.plotfft(3, data, interval);
```

In this statement the third argument, namely `interval`, is of the .NET native type `System.Double`. The builder casts this argument to a MATLAB 1-by-1 double `MWNumericArray` type (which is a wrapper class containing a MATLAB double array).

See "Data Conversion Rules" on page 11-4 for a list of all the data types that are supported along with their equivalent types in the MATLAB product.

---

**Note** There are some data types commonly used in the MATLAB product that are not available as native .NET types. Examples are cell arrays, structure arrays, and arrays of complex numbers. Represent these array types as instances of `MWCellArray`, `MWStructArray`, and `MWNumericArray`, respectively.

---

### Manually Converting Native Data Types to MATLAB Data Types


- "Example: Native Data Conversion" on page 4-12
- "Specifying the Type" on page 4-12
- "Specifying Optional Arguments" on page 4-13
- "Passing a Variable Number of Outputs" on page 4-15

**Example: Native Data Conversion.** The builder provides MATLAB array classes in order to facilitate data conversion between native data and compiled MATLAB functions.

This example explicitly creates a numeric constant using the constructor for the `MWNumericArray` class with a `System.Int32` argument. This variable can then be passed to one of the generated MATLAB Builder NE methods.

```
int data = 24;
MWNumericArray array = new MWNumericArray(data);
Console.WriteLine("Array is of type " + array.NumericType);
```

When you run this example, the results are:

```
Array is of type double
```

In this example, the native integer (`int data`) is converted to an `MWNumericArray` containing a 1-by-1 MATLAB double array, which is the default MATLAB type.

---

**Tip** To preserve the integer type (rather than convert to the default double type), you can use the constructor provided by `MWNumericArray` for this purpose. Preserving the integer type can help to save space.

---

The MATLAB Builder NE product does not support some MATLAB array types because they are not CLS-compliant. See "Unsupported MATLAB Array Types" on page 11-6 for a list of the unsupported types.

For more information about the concepts involved in data conversion, see "Managing Data Conversion Issues with MATLAB® Builder NE Data Conversion Classes" on page 4-9.

**Specifying the Type.** If you want to create a MATLAB numeric array of a specific type, set the optional `makeDouble` argument to `False`. The native type then determines the type of the MATLAB array that is created.

Here, the code specifies that the array should be constructed as a MATLAB 1-by-1 16-bit integer array:

```
short data = 24;
MWNumericArray array = new MWNumericArray(data, false);
Console.WriteLine("Array is of type " + array.NumericType);
```

Running this example produces the following results:

```
Array is of type int16
```

**Specifying Optional Arguments.** In the MATLAB product, `varargin` and `varargout` are used to specify arguments that are not required. Consider the following MATLAB function:

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

This function returns the sum of the inputs. The inputs are provided as a `varargin`, which means that the caller can specify any number of inputs to the function. The result is returned as a scalar `double` array.

For the `mysum` function, the MATLAB Builder NE product generates the following interfaces:

```
// Single output interfaces
public MWArray mysum()
public MWArray mysum(params MWArray[] varargin)
// Standard interface
public MWArray[] mysum(int numArgsOut)
public MWArray[] mysum(int numArgsOut,
           params MWArray[] varargin)
// feval interface
public void mysum(int numArgsOut, ref MWArray ArgsOut,
      params MWArray[] varargin)
```

The `varargin` arguments can be passed as either an `MWArray[]`, or as a list of explicit input arguments. (In C#, the `params` modifier for a method argument specifies that a method accepts any number of parameters of the specific type.) Using `params` allows your code to add any number of optional inputs to the encapsulated MATLAB function.

Here is an example of how you might use the single output interface of the `mysum` method in a .NET application:

```
static void Main(string[] args]
{
MWArray sum= null;
MySumClass mySumClass = null;
try
  {
   mySumClass= new MySumClass();
   sum= mySumClass.mysum((double)2, 4);
   Console.WriteLine("Sum= {0}", sum);
   sum= mySumClass.mysum((double)2, 4, 6, 8);
   Console.WriteLine("Sum= {0}", sum);
  }
}
```

The number of input arguments can vary.

---

**Note** For this particular signature, you must explicitly cast the first argument to MWArray or a type other than integer. Doing this distinguishes the signature from the method signature, which takes an integer as the first argument. If the first argument is not explicitly cast to MWArray or as a type other than integer, the argument can be mistaken as representing the number of output arguments.

---

### Examples of Passing Input Arguments

The following examples show generated code for the `myprimes` MATLAB function, which has the following definition:

```
function p = myprimes(n)
p = primes(n);
```

### Construct a Single Input Argument

The following sample code constructs `data` as a `MWNumericArray`, to be passed as input argument:

```
MWNumericArray data = 5;
MyPrimesClass myClass = new MyPrimesClass();
```

```
MWArray primes = myClass.myprimes(data);
```

### Pass a Native .NET Type

This example passes a native double type to the function.

```
MyPrimesClass myClass = new MyPrimesClass();
MWArray primes = myClass.myprimes((double)13);
```

The input argument is converted to a MATLAB 1-by-1 double array, as required by the MATLAB function. This is the default conversion rule for a native double type (see "Data Conversion Rules" on page 11-4 for a discussion of the default data conversion for all supported .NET types).

### Use the feval Interface

This interface passes both input and output arguments on the right-hand side of the function call. The output argument `primes` must be preceded by a `ref` attribute.

```
MyPrimesClassmyClass = new MyPrimesClass();
MWArray[] maxPrimes = new MWArray[1];
maxPrimes[0] = new MWNumericArray(13);
MWArray[] primes = new MWArray[1];
myClass.myprimes(1, ref primes, maxPrimes);
```

**Passing a Variable Number of Outputs.** When present, `varargout` arguments are handled in the same way that `varargin` arguments are handled. Consider the following MATLAB function:

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

This function returns a list of random `double` vectors such that the length of the `i`th vector is equal to `i`. The builder generates a .NET interface to this function as follows:

```
public void randvectors()
public MWArray[] randvectors(int numArgsOut)
```

```
public void randvectors(int numArgsOut, ref MWArray[] varargout)
```

#### Usage Example

Here, the standard interface is used and two output arguments are requested:

```
MyVarargOutClass myClass = new MyVarargOutClass();
MWArray[] results = myClass.randvectors(2);
Console.WriteLine("First output= {0}", results[0]);
Console.WriteLine("Second output= {0}", results[1]);
```

### Handling Return Values

The previous examples show guidelines to use if you know the type and dimensionality of the output argument. Sometimes, in MATLAB programming, this information is unknown, or can vary. In this case, the code that calls the method might need to query the type and dimensionality of the output arguments.

There are two ways to make the query:

- Use .NET reflection to query any object for its type.
- Use any of several methods provided by the `MWArray` class to query information about the underlying MATLAB array.

**Using .NET Reflection.** You can use *reflection* to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object. You can then invoke the type's methods or access its fields and properties. See the MSDN Library for more information about reflection.

The following code sample calls the `myprimes` method, and then determines the type using reflection. The example assumes that the output is returned as a numeric vector array but the exact numeric type is unknown.

```
public void GetPrimes(int n)
{
 MWArray primes= null;
 MyPrimesClass myPrimesClass= null;
 try
  {
```

```
myPrimesClass= new MyPrimesClass();
primes= myPrimesClass.myprimes((double)n);
Array primesArray= ((MWNumericArray)primes).
 ToVector(MWArrayComponent.Real);
if (primesArray is double[])
 {
 double[] doubleArray= (double[])primesArray;
 /* Do something with doubleArray . . . */
 }
else if (primesArray is float[])
 {
 float[] floatArray= (float[])primesArray;
 /* Do something with floatArray . . . */
 }
else if (primesArray is int[])
 {
 int[] intArray= (int[])primesArray;
 /*Do something with intArray . . . */
 }
else if (primesArray is long[])
 {
 long[] longArray= (long[])primesArray;
 /*Do something with longArray . . . */
 }
else if (primesArray is short[])
 {
 short[] shortArray= (short[])primesArray;
 /*Do something with shortArray . . . */
 }
else if (primesArray is byte[])
 {
 byte[] byteArray= (byte[])primesArray;
 /*Do something with byteArray . . . */
 }
else
 {
 throw new ApplicationException("
  Bad type returned from myprimes");
 }
}
```

```
}
```

The example uses the `toVector` method to return a .NET primitive array (`primesArray`), which represents the underlying MATLAB array. See the following code fragment from the example:

```
primes= myPrimesClass.myprimes((double)n);
  Array primesArray= ((MWNumericArray)primes).
   ToVector(MWArrayComponent.Real);
```

---

**Note** The `toVector` is a method of the `MWNumericArray` class. It returns a copy of the array component in column major order. The type of the array elements is determined by the data type of the numeric array.

---

**Using MWArray Query.** The next example uses the `MWNumericArray` `NumericType` method, along with `MWNumericType` enumeration to determine the type of the underlying MATLAB array. See the `switch (numericType)` statement.

```
public void GetPrimes(int n)
{
 MWArray primes= null;
 MyPrimesClass myPrimesClass= null;
 try
  {
  myPrimesClass= new MyPrimesClass();
  primes= myPrimesClass.myprimes((double)n);
  if ((!primes.IsNumericArray) || (2 !=
   primes.NumberofDimensions))
   {
   throw new ApplicationException("Bad type returned
    by mwprimes");
   }
  MWNumericArray _primes= (MWNumericArray)primes;
  MWNumericType numericType= _primes.NumericType;
  Array primesArray= _primes.ToVector(
   MWArrayComponent.Real);
  switch (numericType)
```

```
{
case MWNumericType.Double:
 {
 double[] doubleArray= (double[])primesArray;
 /* (Do something with doubleArray . . .) */
 break;
 }
case MWNumericType.Single:
 {
 float[] floatArray= (float[])primesArray;
 /* (Do something with floatArray . . .) */
 break;
 }
case MWNumericType.Int32:
 {
 int[] intArray= (int[])primesArray;
 /* (Do something with intArray . . .) */
 break;
 }
case MWNumericType.Int64:
 {
 long[] longArray= (long[])primesArray;
 /* (Do something with longArray . . .) */
 break;
 }
case MWNumericType.Int16:
 {
 short[] shortArray= (short[])primesArray;
 /* (Do something with shortArray . . .) */
 break;
 }
case MWNumericType.UInt8:
 {
 byte[] byteArray= (byte[])primesArray;
 /* (Do something with byteArray . . .) */
 break;
 }
default:
 {
 throw new ApplicationException("Bad type returned
```

```
        by myprimes");
       }
      }
     }
    }
```

The code in the example also checks the dimensionality by calling `NumberOfDimensions`; see the following code fragment:

```
if ((!primes.IsNumericArray) || (2 !=
   primes.NumberofDimensions))
   {
   throw new ApplicationException("Bad type returned
    by mwprimes");
      }
```

This call throws an exception if the array is not numeric and of the proper dimension.

## Using MATLAB API Functions in a C# Program


- "Overview" on page 4-20
- "Example: Using functions engOpen and engEvalString from the MATLAB Engine API in a C# Program" on page 4-20


### Overview

You include functions from MATLAB APIs, such as the Engine API, in your C# code by using the `DllImport` attribute to import functions from `libeng.dll` (written in unmanaged C) and then declaring those functions as C# equivalents. The imported Engine functions are called using the `P/Invoke` mechanism, as illustrated in the example below.

### Example: Using functions engOpen and engEvalString from the MATLAB Engine API in a C# Program

1. Open Microsoft Visual Studio .NET.
2. Select **File > New > Project**.

**3** Select **Visual C# Applications** in the left pane and **Console Application** in the right pane. Click **OK**.

**4** Auto-generated code appears. Replace the auto-generated code with this code and run:

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace ConsoleApplication8
{
    class MatlabEng
    {
        [DllImport("libeng.dll")]
        static extern IntPtr engOpen(string startcmd);

        [DllImport("libeng.dll")]
        static extern IntPtr engEvalString(IntPtr engine,
                                           string Input);

        public MatlabEng()
        {
            IntPtr engine;
            engine = engOpen(null);
            if (engine == IntPtr.Zero)
                throw new NullReferenceException("Failed to
                                    Initialize Engine");

            engEvalString(engine, "surf(peaks)");
        }

        ~MatlabEng()
        {
        }
    }


    class StartProg
    {
```

```
            public static void Main()
            {
               MatlabEng mat = new MatlabEng();
            }
        }
    }
```

## Accessing Real or Imaginary Components Within Complex Arrays


- "Extracting Real or Imaginary Components" on page 4-22
- "Returning Values with Component Indexing" on page 4-22
- "Assigning Values with Component Indexing" on page 4-23
- "Converting MATLAB Arrays to .NET Arrays Using Component Indexing" on page 4-23


### Extracting Real or Imaginary Components

When you access a complex array (an array made up of both real and imaginary data), you extract both real and imaginary parts (called components) by default. This method call, for example, extracts both real and imaginary components:

```
MWNumericArray complexResult= complexDouble[1, 2];
```

It is also possible, when calling a method to return or assign a value, to extract only the real or imaginary component of a complex matrix. To do this, call the appropriate *component indexing* method.

This section describes how to use component indexing when returning or assigning a value, and also describes how to use component indexing to convert MATLAB arrays to .NET arrays using the `ToArray` or `ToVector` methods.

### Returning Values with Component Indexing

The following section illustrates how to return values from full and sparse arrays using component indexing.

**Implementing Component Indexing on Full Complex Numeric Arrays.** To return the real or imaginary component from a full complex numeric array, call the `.real` or `.imaginary` method on `MWArrayComponent` as follows:

```
complexResult= complexDouble[MWArrayComponent.Real, 1, 2];
complexResult= complexDouble[MWArrayComponent.Imaginary, 1, 2];
```

**Implementing Component Indexing on Sparse Complex Numeric Arrays (Microsoft Visual Studio 8 and Later).** To return the real or imaginary component of a sparse complex numeric array, call the `.real` or `.imaginary` method `MWArrayComponent` as follows:

```
complexResult= sparseComplexDouble[MWArrayComponent.Real, 4, 3];
complexResult = sparseComplexDouble[MWArrayComponent.Imaginary, 4, 3];
```

## Assigning Values with Component Indexing

The following section illustrates how to assign values to full and sparse arrays using component indexing.

**Implementing Component Indexing on Full Complex Numeric Arrays.** To assign the real or imaginary component to a full complex numeric array, call the `.real` or `.imaginary` method `MWArrayComponent` as follows:

```
matrix[MWArrayComponent.Real, 2, 2]= 5;
matrix[MWArrayComponent.Imaginary, 2, 2]= 7:
```

## Converting MATLAB Arrays to .NET Arrays Using Component Indexing

The following section illustrates how to use the `ToArray` and `ToVector` methods to convert full and sparse MATLAB arrays and vectors to .NET arrays and vectors respectively.

**Converting MATLAB Arrays to .NET Arrays.** To convert MATLAB arrays to .NET arrays call the `toArray` method with either the `.real` or `.imaginary` method, as needed, on `MWArrayComponent` as follows:

```
Array nativeArray_real= matrix.ToArray(MWArrayComponent.Real);
```

```
Array nativeArray_imag= matrix.ToArray(MWArrayComponent.Imaginary);
```

**Converting MATLAB Arrays to .NET Vectors.** To convert MATLAB vectors to .NET vectors (single dimension arrays) call the `.real` or `.imaginary` method, as needed, on `MWArrayComponent` as follows:

```
Array nativeArray= sparseMatrix.ToVector(MWArrayComponent.Real);
Array nativeArray= sparseMatrix.ToVector(MWArrayComponent.Imaginary);
```

## Adding Fields to Data Structures and Data Structure Arrays

When adding fields to data structures and data structure arrays, do so using standard programming techniques. Do not use the `set` command as a shortcut.

For examples of how to correctly add fields to data structures and data structure arrays, see the programming examples in Chapter 5, “Sample Applications (C#)” and Chapter 6, “Sample Applications (Microsoft® Visual Basic .NET)”.

## Using MATLAB Array Indexing

.NET Builder provides indexers to support a subset of MATLAB array indexing.

---

**Note** If each element in a large array returned by a .NET Builder component is to be indexed, the returned MATLAB array should first be converted to a native array using the `toArray()` method. This results in much better performance.

---

Don’t keep the array in MATLAB type; convert it to a native array first. See Chapter 1, “Getting Started” for an example of native type conversion.

## Blocking Execution of a Console Application that Creates Figures


- "WaitForFiguresToDie Method" on page 4-25
- "Code Fragment: Using WaitForFiguresToDie to Block Execution" on page 4-26


### WaitForFiguresToDie Method

The MATLAB Builder NE product adds a `WaitForFiguresToDie` method to each .NET class that it creates. `WaitForFiguresToDie` takes no arguments. Your application can call `WaitForFiguresToDie` any time during execution.

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

- There are one or more figures open that were created by a .NET component created by the builder.
- The method that displays the graphics requires user input before continuing.
- The method that calls the figures was called from `main()` in a console program.

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

---

**Tip** Consider using the `console.readline` method when possible as it accomplishes much of this functionality in a standardized manner.

---

**Caution** Use care when calling the `WaitForFiguresToDie` method. Calling this method from an interactive program, such as Microsoft Excel, can hang the application. This method should be called *only* from console-based programs.

---

## Code Fragment: Using WaitForFiguresToDie to Block Execution

The following example illustrates using `WaitForFiguresToDie` from a .NET application. The example uses a .NET component created by the MATLAB Builder NE product; the object encapsulates MATLAB code that draws a simple plot.

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

2. In this folder, create the following MATLAB file:

```
drawplot.m

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

3. Use MATLAB Builder NE to create a .NET component with the following properties:

| | |
|---|---|
| Component name | `Figure` |
| Class name | `Plotter` |

4. Create a .NET program in a file named `runplot` with the following code:

```
using Figure.Plotter;

public class Main {
  public static void main(String[] args) {
    try {
      plotter p = new Plotter();
      try {
        p.showPlot();
        p.WaitForFiguresToDie();
      }
     catch (Exception e) {
      console.writeline(e);
              }
    }
       }
  }
```

**5** Compile the application.

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

---

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

---

## Handling Errors

As with managed code, any errors that occur during execution of an MATLAB function or during data conversion are signaled by a standard .NET exception.

Like any other .NET application, an application that calls a method generated by the MATLAB Builder NE product can handle errors by either

- Catching and handling the exception locally
- Allowing the calling method to catch it

Here are examples for each way of handling errors.

In the `GetPrimes` example the method itself handles the exception.

```
public double[] GetPrimes(int n)
{
 MWArray primes= null;
 MyPrimesClass myPrimesClass= null;
 try
  {
   myPrimesClass= new MyPrimesClass();
   primes= myPrimesClass.myprimes((double)n);
   return (double[])(MWNumericArray)primes).
        ToVector(MWArrayComponent.Real);
  }
 catch (Exception ex)
  {
  Console.WriteLine("Exception: {0}", ex);
```

```
    return new double[0];
    }
  }
```

In the next example, the method that calls `myprimes` does not catch the exception. Instead, its calling method (that is, the method that calls the method that calls `myprimes`) handles the exception.

```
public double[] GetPrimes(int n)
{
   MWArray primes= null;
   MyPrimesClass myPrimesClass= null;
   try
     {
        myPrimesClass= new MyPrimesClass();
        primes= myPrimesClass.myprimes((double)n);
        return (double[])(MWNumericArray)primes).
        ToVector(MWArrayComponent.Real);
     }

   catch (Exception e)
     {
        throw;
     }
}
```

## Using Dispose to Explicitly Free Resources

---

**Note** As of R2009b, native memory management for `mxArray` is automatically handled by .NET's CLR memory manager. There is no longer a reason to manually disable native memory management when working with `mxArray`. Calls to disable memory management will result in a null operation.

---

Usually the `Dispose` method is called from a `finally` section in a `try-finally` block as you can see in the following example:

```
try
 {
   /* Allocate a huge array */
```

```
     MWNumericArray array = new MWNumericArray(1000,1000);
    .
    .  (use the array)
    .
   }
  finally
   {
     /* Explicitly dispose of the managed array and its */
     /* native resources */
   if (null != array)
    {
      array.Dispose();
    }
   }
```

The statement `array.Dispose()` frees the memory allocated by both the managed wrapper and the native MATLAB array.

The `MWArray` class provides two disposal methods: `Dispose` and the static method `DisposeArray`. The `DisposeArray` method is more general in that it disposes of either a single `MWArray` or an array of arrays of type `MWArray`.

# Deploying Your .NET Component to End Users

| In this section… |
|---|
| "Dynamically Specifying Run-Time Options to the MCR" on page 4-30 |
| "Extracting the CTF Archive Manually Using the MCR Component Cache" on page 4-32 |
| "Improving Data Access Using the MCR User Data Interface and MATLAB® Builder NE" on page 4-33 |
| "Using Enhanced XML Documentation Files" on page 4-38 |

## Dynamically Specifying Run-Time Options to the MCR

- "What Run-Time Options Can You Specify?" on page 4-30
- "Getting MCR Option Values Using MWMCR" on page 4-30

### What Run-Time Options Can You Specify?

As of R2009a, you can pass MCR run-time options `-nojvm` and `-logfile` to MATLAB Builder NE from a client application using the assembly-level attributes `NOJVM` and `LOGFILE`. You retrieve values of these attributes by calling methods of the `MWMCR` class to access MCR attributes and MCR state.

### Getting MCR Option Values Using MWMCR

The `MWMCR` class provides several methods to get MCR option values. The following table lists methods supported by this class.

| MWMCR Method | Purpose |
|---|---|
| `MWMCR.IsMCRInitialized()` | Returns `true` if MCR is initialized, otherwise returns `false`. |
| `MWMCR.IsMCRJVMEnabled()` | Returns `true` if MCR is launched with .NET Virtual Machine (JVM), otherwise returns `false`. |
| `MWMCR.GetMCRLogFileName()` | Returns the name of the log file passed with the `LOGFILE` attribute. |

**Default MCR Options .** If you pass no MCR options (you provide no attributes), the MCR is launched with default option values:

| MCR Run-Time Option | Default Option Values |
|---|---|
| .NET Virtual Machine (JVM) | `NOJVM(false)` |
| Log file usage | `LOGFILE(null)` |

These options are all write-once, read-only properties.

Use the following attributes to represent the MCR options you want to modify.

| MWMCR Attribute | Purpose |
|---|---|
| `NOJVM` | Lets users launch MCR with or without a JVM. It takes a Boolean as input. For example, `NOJVM(true)` launches MCR without a JVM. |
| `LOGFILE` | Lets users pass the name of a log file, taking the file name as input. For example, `LOGFILE("`*`logfile3.txt`*`")`. |

## Example: Passing MCR Option Values from a C# Application

Following is an example of how MCR option values are passed from a client-side C# application:

```
[assembly: NOJVM(false), LOGFILE("logfile3.txt")]
   namespace App1
   {
       class Program
       {
           static void Main(string[] args)
           {
               Console.WriteLine("In side main...");
               try
               {
```

```
                    myclass cls = new myclass();
                    cls.hello();
                    Console.WriteLine("Done!!");
                    Console.ReadLine();
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
```

## Extracting the CTF Archive Manually Using the MCR Component Cache


- "Requirements for CTF Archive Extraction" on page 4-32
- "Why Would You Want to Extract the CTF Archive?" on page 4-32
- "How Do I Extract the CTF Archive?" on page 4-33


As of R2008b, CTF data is automatically embedded directly in .NET and COM components by default.

### Requirements for CTF Archive Extraction

In order to extract the CTF archive manually, you must build the component using the `mcc -C` option.

### Why Would You Want to Extract the CTF Archive?

Some of the reasons you would choose to override the default CTF archive embedding behavior include:

- You need to specifically define the location where you want the CTF archive to be extracted
- You want to add diagnostic error printing options that can be used when extracting the CTF, for troubleshooting purposes
- You want to tune the MCR component cache size for performance reasons.

### How Do I Extract the CTF Archive?

To extract the CTF archive, use these environment variables to define the following settings.

| Environment Variable | Purpose | Notes |
|---|---|---|
| MCR_CACHE_ROOT | When set to the location of where you want the CTF 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 CTF 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 `.max_size`, which resides in the file designated by running the `mcrcachedir` command, with the desired cache size limit. |

## Improving Data Access Using the MCR User Data Interface and MATLAB Builder NE

This feature 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 configuration information to a client running an application created with the Parallel Computing Toolbox™ software. Configuration information may be supplied (and changed) on a per-execution basis. For example, two instances of the same application may run simultaneously with different configuration files.
- 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 NE software supports a per-MCR instance state access through an object-oriented API. Unlike MATLAB Compiler, access to a 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. Alternatively, you can use a helper function to call these methods as shown in "Example: Supplying Run-Time Configuration Information for Parallel Computing Toolbox Applications" on page 4-34.

For more information, see"Improving Data Access Using the MCR User Data Interface" in the MATLAB Compiler User's Guide.

### Example: Supplying Run-Time Configuration 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 configuration `.mat` file for Parallel Computing Toolbox applications.

#### Step 1: Write Your PCT Code.

1. Compile `sample_pct.m` in MATLAB. By default, the code uses the local scheduler, starts the workers, and evaluates the result.

```
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;
matlabpool('open',4);
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']);
matlabpool('close','force');
disp('done');
speedup = (time1/time2);
```

**2** Run the code as follows:

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

**3** Verify that you get the following results;

```
Starting matlabpool using the 'local'
                 configuration ... 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.
Did not find any pre-existing parallel jobs created
by matlabpool.
done
a =
   1.7643
```

**Step 2: Set the Parallel Computing Toolbox Configuration.** In order to compile MATLAB code to a .NET component and utilize the Parallel Computing Toolbox, the `mcruserdata` must be set directly from MATLAB. There is no .NET 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 .NET class. This is a separate MATLAB function that uses `setmcruserdata` to set the Parallel Computing Toolbox configuration 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 Configuration file:
if(isdeployed)
    [matfile, matpath,c] = uigetfile('*.mat');
                          % let the USER select file
    setmcruserdata('ParallelConfigurationFile',
                              [matpath matfile]);
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 -W 'dotnet:netPctComp,NetPctClass'
        init_sample_pct.m sample_pct.m -T link:lib
```

Alternately, you can use the Deployment Tool as follows:

1. Follow the steps in "Building Your Component" on page 1-13 to compile your application. When the compilation finishes, a new folder (with the same name as the project) is created. This folder contains two subfolders: `distrib` and `src`.

| | |
|---|---|
| Project Name | `netPctComp` |
| Class Name | `NetPctClass` |
| File to Compile | `sample_pct.m` and `init_sample_pct.m` |

**2** To deploy the compiled application, copy the `distrib` folder, which contains the following, to your end users. The packaging function of `deploytool` offers a convenient way to do this.

- `netPctComp.dll`
- `MWArray.dll`
- MCR installer
- MAT file containing cluster configuration information

---

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

---

**Step 4: Write the .NET Driver Application.** After adding references to your component and to `MWArray` in your Microsoft Visual Studio project: write the following .NET driver application to use the component, as follows. See "Creating a Reference to Your Component in C#/.NET Code" on page 1-26 and "Creating a Reference to the MWArray API" on page 1-27 in the Chapter 1, "Getting Started" chapter of this User's Guide for more information.

---

**Note** This example code was written using Microsoft Visual Studio 2008.

---

```
using System;
using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;
using netPctComp;
namespace PctNet
{
    class Program
    {
        static void Main(string[] args)
```

```
            {
                try
                {
                    NetPctClass A = new NetPctClass();
                    // Initialize the PCT set up
                    A.init_sample_pct();
                    double var = 300;
                    MWNumericArray  out1;
                    MWNumericArray in1 = new MWNumericArray(300);
                    out1 = (MWNumericArray)A.sample_pct(in1);
                    Console.WriteLine("The speedup is {0}", out1);
                    Console.ReadLine();
                            // Wait for user to exit application
                }
                catch (Exception exception)
                {
                    Console.WriteLine("Error: {0}", exception);
                }

            }
        }
    }
```

The output is as follows:

```
file:///C:/pct_compile/builderNE/deploy/PctNet/bin/Debug/PctNet.EXE
The speedup is 2.1565
```

## Using Enhanced XML Documentation Files

Every MATLAB® Builder NE component includes a `readme.txt` file in the `src` and `distrib` directories. This file outlines the contents of auto-generated documentation templates included with your built component. The documentation templates are HTML and XML files that can be read and processed by any number of third-party tools.

- `MWArray.xml` — This file describes the `MWArray` data conversion classes and their associated methods. Documentation for `MWArray` classes and their methods are available here.

- *component_name*.xml — This file contains the code comments for your component. Using a third party documentation tool, you can combine this file with MWArray.xml to produce a complete documentation file that can be packaged with the component assembly for distribution to end users.
- *component_name*_overview.html — Optionally include this file in the generated documentation file. It contains an overview of the steps needed to access the component and how to use the data conversion classes, contained in the MWArray class hierarchy, to pass arguments to the generated component and return the results.

# Accessing Your Component On Another Computer

To implement your .NET component on a computer other than the one on which it was built:

1. If the component is not already installed on the machine where you want to develop your application, run the self-extracting executable that you created in "Deploying a Component Using the Magic Square Example" on page 1-8.

   This step is not necessary if you are developing your application on the same machine where you created the .NET component.

2. Reference the .NET component in your Microsoft Visual Studio project or from the command line of a CLS-compliant compiler.

   You must also add a reference to the `MWArray` component in *matlabroot*`\toolbox\dotnetbuilder\bin\`*architecture*`\`*framework_version*. See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

3. Instantiate the generated .NET Builder classes and call the class methods as you would with any .NET class. To marshal data between the native .NET types and the MATLAB array type, you need to use either the `MWArray` data conversion classes or the `MWArray` native API. See MWArray Class Library Reference (available online only) for details about the `MWArray` API for this class library.

# For More Information

| If you want to... | See... |
| --- | --- |
| Learn how to build a component and perform basic integration tasks using C# code | Chapter 1, "Getting Started" |
| • Basic MATLAB Programmer tasks<br>• How the deployment products process your MATLAB functions<br>• How the deployment products work together | Chapter 2, "Writing Deployable MATLAB Code" |
| Learn about supported MATLAB Builder NE targets | Chapter 3, "Building Your .NET Component" |
| Building your component and using the Deployment Tool with the command line option | Chapter 3, "Building Your .NET Component" |

# 5

# Sample Applications (C#)

**Note** The examples for the MATLAB Builder NE product are in *matlabroot*\toolbox\dotnetbuilder\Examples\VS*versionnumber*, where *matlabroot* is the folder where the MATLAB product is installed and VS*versionnumber* specifies the version of Microsoft Visual Studio .NET you are using (in this case VS8 ). If you have Microsoft Visual Studio .NET installed, you can load projects for all the examples by opening the solution *matlabroot*\toolbox\dotnetbuilder\Examples\VS*versionnumber*\NET\DotNetExamples.sln.


- "Simple Plot Example" on page 5-2
- "Passing Variable Arguments" on page 5-7
- "Spectral Analysis Example" on page 5-13
- "Matrix Math Example" on page 5-20
- "Phonebook Example" on page 5-28


**Note** In addition to these examples, see "Deploying a Component Using the Magic Square Example" on page 1-8 for a simple example that gets you started using the MATLAB Builder NE product.

# Simple Plot Example

| In this section... |
|---|
| "Purpose" on page 5-2 |
| "Procedure" on page 5-2 |

## Purpose

The `drawgraph` function displays a plot of input parameters `x` and `y`. The purpose of the example is to show you how to:

- Use the MATLAB Builder NE product to convert a MATLAB function (`drawgraph`) to a method of a .NET class (`Plotter`) and wrap the class in a .NET component (`PlotComp`).
- Access the component in a C# application (`PlotApp.cs`) by instantiating the `Plotter` class and using the MWArray class library to handle data conversion.

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- Build and run the `PlotCSApp` application, using the Visual Studio .NET development environment.

## Procedure

**1** If you have not already done so, copy the files for this example as follows:

**a** Copy the following folder that ships with the MATLAB product to your work folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\PlotExample
```

**b** At the MATLAB command prompt, change folder to the new `PlotExample\PlotComp` subfolder in your work folder.

**2** Write the `drawgraph` function as you would any MATLAB function.

This code is already in your work folder in `PlotExample\PlotComp\drawgraph.m`.

**3** While in MATLAB, issue the following command to open the Deployment Tool window:

```
deploytool
```

**4** Build the .NET component. See the instructions in “Building Your Component” on page 1-13 for more details. Use the following information:

| | |
|---|---|
| Project Name | `PlotComp` |
| Class Name | `Plotter` |
| File to compile | `drawgraph.m` |

**5** Write source code for a C# application that accesses the component.

The sample application for this example is in *`matlabroot`*`\toolbox\dotnetbuilder\Examples\VS8\PlotExample\PlotCSApp\PlotApp.cs`.

The program listing is shown here.

### PlotApp.cs

```
// *******************************************************************************
//
// PlotApp.cs
//
// This example demonstrates how to use MATLAB Builder NE to build a component
// that displays a MATLAB figure window.
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *******************************************************************************

using System;
```

```
using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using PlotComp;

namespace MathWorks.Examples.PlotApp
 {
  /// <summary>
  /// This application demonstrates plotting x-y data by graphing a simple
  /// parabola into a MATLAB figure window.
  /// </summary>
  class PlotCSApp
    {
      #region MAIN

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
        {
          try
            {
              const int numPoints= 10;  // Number of points to plot

              // Allocate native array for plot values
              double [,] plotValues= new double[2, numPoints];

              // Plot 5x vs x^2
              for (int x= 1; x <= numPoints; x++)
                {
                  plotValues[0, x-1]= x*5;
                  plotValues[1, x-1]= x*x;
                }

              // Create a new plotter object
              Plotter plotter= new Plotter();

              // Plot the two sets of values - Note the ability to cast
```

```
                    the native array to a MATLAB numeric array
            plotter.drawgraph((MWNumericArray)plotValues);

            Console.ReadLine();  // Wait for user to exit application
          }

        catch(Exception exception)
          {
            Console.WriteLine("Error: {0}", exception);
          }
      }

    #endregion
  }
}
```

The program does the following:

- Creates two arrays of double values
- Creates a `Plotter` object.
- Calls the `drawgraph` method to plot the equation using the MATLAB `plot` function.
- Uses `MWNumericArray` to represent the data needed by the `drawgraph` method to plot the equation.
- Uses a `try-catch` block to catch and handle any exceptions.

The statement

```
Plotter plotter= new Plotter();
```

creates an instance of the `Plotter` class, and the statement

```
plotter.drawgraph((MWNumericArray)plotValues);
```

explicitly casts the native `plotValues` to `MWNumericArray` and then calls the method `drawgraph`.

6. Build the `PlotCSApp` application using Visual Studio .NET.

   a. The `PlotCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PlotCSApp.csproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **PlotCSApp.csproj > Open Outside MATLAB**.

   b. Add a reference to the `MWArray` component, which is *`matlabroot`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`*`\mwarray.dll`. See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

   c. Add or, if necessary, fix the location of a reference to the `PlotComp` component which you built in a previous step. (The component, `PlotComp.dll`, is in the `\PlotExample\PlotComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

7. Build and run the application in Visual Studio .NET.

# Passing Variable Arguments

**Note** This example is similar to "Simple Plot Example" on page 5-2, except that the MATLAB function to be encapsulated takes a variable number of arguments instead of just one.

The purpose of the example is to show you the following:

- How to use the builder to convert a MATLAB function, `drawgraph`, which takes a variable number of arguments, to a method of a .NET class (`Plotter`) and wrap the class in a .NET component (`VarArgComp`). The `drawgraph` function (which can be called as a method of the `Plotter` class) displays a plot of the input parameters.
- How to access the component in a C# application (`VarArgApp.cs`) by instantiating the `Plotter` class and using `MWArray` to represent data.

  **Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

- How to build and run the `VarArgDemoApp` application, using the Visual Studio .NET development environment.

### Step-by-Step Procedure

**1** If you have not already done so, copy the files for this example as follows:

  **a** Copy the following folder that ships with the MATLAB product to your work folder:

  ```
  matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\VarArgExample
  ```

  **b** At the MATLAB command prompt, `cd` to the new `VarArgExample` subfolder in your work folder.

**2** Write the MATLAB functions as you would any MATLAB function.

The code for the functions in this example is as follows:

## drawgraph.m

```
function [xyCoords] = DrawGraph(colorSpec, varargin)
...
    numVarArgIn= length(varargin);
    xyCoords= zeros(numVarArgIn, 2);

    for idx = 1:numVarArgIn
        xCoord = varargin{idx}(1);
        yCoord = varargin{idx}(2);

        x(idx) = xCoord;
        y(idx) = yCoord;

        xyCoords(idx,1) = xCoord;
        xyCoords(idx,2) = yCoord;
    end

    xmin = min(0, min(x));
    ymin = min(0, min(y));

    axis([xmin fix(max(x))+3 ymin fix(max(y))+3])

    plot(x, y, 'color', colorSpec);
```

## extractcoords.m

```
function [varargout] = ExtractCoords(coords)
%EXTRACTCOORDS Extracts a variable number of two element x and y
% coordinate vectors from a two column array
%   [VARARGOUT] = EXTRACTCOORDS(COORDS) Extracts x,y coordinates
$   from a two column array
%   This file is used as an example for the .NET Builder
%   Language product.

%   Copyright 2001-2010 The MathWorks, Inc.
```

```
%   $Revision: 1.1.4.58.6.4 $  $Date: 2010/07/14 16:41:26 $

    for idx = 1:nargout
        varargout{idx}= coords(idx,:);
    end
```

This code is already in your work folder in `/VarArgExample/VarArgComp/`.

3 While in MATLAB, issue the following command to open the Deployment Tool dialog box:

```
deploytool
```

4 Build the .NET component. See the instructions in "Building Your Component" on page 1-13 for more details. Use the following information:

| | |
|---|---|
| Project Name | `VarArgComp` |
| Class Name | `Plotter` |
| File to compile | `extractcoords.m  drawgraph.m` |

5 Write source code for an application that accesses the component.

The sample application for this example is in `VarArgExample\VarArgCSApp\VarArgApp.cs`.

The program listing is shown here.

**VarArgApp.cs**

```
// *******************************************************************************
//
//VarArgApp.cs
//
// This example demonstrates how to use MATLAB Builder NE to build a component
// with a variable number of input and output arguments.
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *******************************************************************************
```

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using VarArgComp;


namespace MathWorks.Examples.VarArgApp
 {
  /// <summary>
  /// This application demonstrates how to call components
  ///  having methods with varargin/vargout arguments.
  /// </summary>
  class VarArgApp
    {
      #region MAIN

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
        {
          // Initialize the input data
          MWNumericArray colorSpec= new double[]
                                  {0.9, 0.0, 0.0};
          MWNumericArray data=
                  new MWNumericArray(new int[,]{{1,2},{2,4},
                                 {3,6},{4,8},{5,10}});
          MWArray[] coords= null;

          try
            {
              // Create a new plotter object
              Plotter plotter= new Plotter();

              //Extract a variable number of two element x and y coordinate
              //  vectors from the data array
```

```
          coords= plotter.extractcoords(5, data);

          // Draw a graph using the specified color to connect the
          //   variable number of input coordinates.
          // Return a two column data array containing the input coordinates.
          data= (MWNumericArray)plotter.drawgraph(colorSpec,
            coords[0], coords[1], coords[2],coords[3], coords[4]);

          Console.WriteLine("result=\n{0}", data);

          Console.ReadLine();  // Wait for user to exit application

          // Note: You can also pass in the coordinate array directly.
          data= (MWNumericArray)plotter.drawgraph(colorSpec, coords);

          Console.WriteLine("result=\n{0}", data);

          Console.ReadLine();  // Wait for user to exit application
        }

      catch(Exception exception)
        {
          Console.WriteLine("Error: {0}", exception);
        }
    }

  #endregion
  }
}
```

The program does the following:

- Initializes three arrays (`colorSpec`, `data`, and `coords`) using the `MWArray` class library
- Creates a `Plotter` object
- Calls the `extracoords` and `drawgraph` methods
- Uses `MWNumericArray` to represent the data needed by the methods
- Uses a `try-catch` block to catch and handle any exceptions

The following statements are alternative ways to call the `drawgraph` method:

```
data= (MWNumericArray)plotter.drawgraph(colorSpec,
                coords[0], coords[1], coords[2],coords[3], coords[4]);
...
data= (MWNumericArray)plotter.drawgraph((MWArray)colorSpec, coords);
```

6 Build the `VarArgApp` application using Visual Studio .NET.

   a The `VarArgCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `VarArgCSApp.csproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **VarArgCSApp.csproj > Open Outside MATLAB**.

   b Add a reference to the `MWArray` component, which is *`matlabroot`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`* `\mwarray.dll.` See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

   c Add or, if necessary, fix the location of a reference to the `VarArgComp` component which you built in a previous step. (The component, `VarArgComp.dll`, is in the `\VarArgExample\VarArgComp\x86\v2.0\debug\distrib` subfolder of your work area.)

7 Build and run the application in Visual Studio .NET.

# Spectral Analysis Example

**In this section...**

"Purpose" on page 5-13

"Procedure" on page 5-15

## Purpose

The purpose of the example is to show you the following:

- How to use the MATLAB Builder NE product to create a component (`SpectraComp`) containing more than one class
- How to access the component in a C# application (`SpectraApp.cs`), including use of the `MWArray` class hierarchy to represent data

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- How to build and run the application, using the Visual Studio .NET development environment

The component `SpectraComp` analyzes a signal and graphs the result. The class, `SignalAnalyzer`, performs a fast Fourier transform (FFT) on an input data array. A method of this class, `computefft`, returns the results of that FFT as two output arrays—an array of frequency points and the power spectral density. The second class, `Plotter`, graphs the returned data using the `plotfft` method. These two methods, `computefft` and `plotfft`, encapsulate MATLAB functions.

The `computefft` method computes the FFT and power spectral density of the input data and computes a vector of frequency points based on the length of the data entered and the sampling interval. The `plotfft` method plots the FFT data and the power spectral density in a MATLAB figure window. The MATLAB code for these two methods resides in two MATLAB files, `computefft.m` and `plotfft.m`, which can be found in:

*matlabroot*\toolbox\dotnetbuilder\Examples\VS8\NET\SpectraExample\SpectraComp

### computefft.m

```
function [fftData, freq, powerSpect] = ComputeFFT(data, interval)
%COMPUTEFFT Computes the FFT and power spectral density.
%   [FFTDATA, FREQ, POWERSPECT] = COMPUTEFFT(DATA, INTERVAL)
%   Computes the FFT and power spectral density of the input data.
%   This file is used as an example for the .NET Builder
%   Language product.
%   Copyright 2001-2010 The MathWorks, Inc.
if (isempty(data))
   fftdata = [];
   freq = [];
   powerspect = [];
   return;
end
if (interval <= 0)
   error('Sampling interval must be greater then zero');
   return;
end
fftData = fft(data);
freq = (0:length(fftData)-1)/(length(fftData)*interval);
powerSpect = abs(fftData)/(sqrt(length(fftData)));
```

### plotfft.m

```
function PlotFFT(fftData, freq, powerSpect)
%PLOTFFT Computes and plots the FFT and power spectral density.
%   [FFTDATA, FREQ, POWERSPECT] = PLOTFFT(DATA, INTERVAL)
%   Computes the FFT and power spectral density of the input data.
%   This file is used as an example for the .NET Builder
%   Language product.
%   Copyright 2001-2010 The MathWorks, Inc.
len = length(fftData);
    if (len <= 0)
        return;
    end
    plot(freq(1:floor(len/2)), powerSpect(1:floor(len/2)))
    xlabel('Frequency (Hz)'), grid on
```

```
title('Power spectral density')
```

## Procedure

**1** If you have not already done so, copy the files for this example as follows:

**a** Copy the following folder that ships with the MATLAB product to your work folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\SpectraExample
```

**b** At the MATLAB command prompt, `cd` to the new `SpectraExample` subfolder in your work folder.

**2** Write the MATLAB code that you want to access.

This example uses `computefft.m` and `plotfft.m`, which are already in your work folder in `SpectraExample\SpectraComp`.

**3** While in MATLAB, issue the following command to open the Deployment Tool window:

```
deploytool
```

**4** Build the .NET component. See the instructions in “Building Your Component” on page 1-13 for more details. Use the following information:

| | |
|---|---|
| Project Name | `SpectraComp` |
| Class Names | `Plotter  SignalAnalyzer` |
| Files to compile | `computefft.m  plotfft.m` |

**5** Write source code for an application that accesses the component.

The sample application for this example is in `SpectraExample\SpectraCSApp\SpectraApp.cs`.

The program listing is shown here.

### SpectraApp.cs

```
// *******************************************************************************
//
//SpectraApp.cs
//
// This example demonstrates how to use MATLAB Builder NE to build a component
// with multiple classes.
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *******************************************************************************

using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using SpectraComp;

namespace MathWorks.Examples.SpectraApp
{
  /// <summary>
  /// This application computes and plots the power spectral density of an input signal.
  /// </summary>
  class SpectraCSApp
    {
      #region MAIN

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
        {
          try
            {
              const double interval= 0.01;  // The sampling interval
              const int numSamples= 1001;   // The number of samples
```

```
   // Construct input data as sin(2*PI*15*t) + (sin(2*PI*40*t) plus a
   // random signal.  Duration= 10; Sampling interval= 0.01
   MWNumericArray data= new MWNumericArray(MWArrayComplexity.Real, MWNumericType.Double, numSamples);

   Random random= new Random();

   // Initialize data
   for (int idx= 1; idx <= numSamples; idx++)
     {
       double t= (idx-1)* interval;

       data[idx]= Math.Sin(2.0*Math.PI*15.0*t) + Math.Sin(2.0*Math.PI*40.0*t) + random.NextDouble();
     }

   // Create a new signal analyzer object
   SignalAnalyzer signalAnalyzer= new SignalAnalyzer();

   // Compute the fft and power spectral density for the data array
   MWArray[] argsOut= signalAnalyzer.computefft(3, data, interval);

   // Print the first twenty elements of each result array
   int numElements= 20;

   MWNumericArray resultArray= new MWNumericArray(MWArrayComplexity.Complex, MWNumericType.Double, numElements);

for (int idx= 1; idx <= numElements; idx++)
  {
   resultArray[idx]= ((MWNumericArray)argsOut[0])[idx];
  }

Console.WriteLine("FFT:\n{0}\n", resultArray);

   for (int idx= 1; idx <= numElements; idx++)
     {
       resultArray[idx]= ((MWNumericArray)argsOut[1])[idx];
     }

   Console.WriteLine("Frequency:\n{0}\n", resultArray);

   for (int idx= 1; idx <= numElements; idx++)
```

```
            {
              resultArray[idx]= ((MWNumericArray)argsOut[2])[idx];
            }

          Console.WriteLine("Power Spectral Density:\n{0}", resultArray);

          // Create a new plotter object
          Plotter plotter= new Plotter();

          // Plot the fft and power spectral density for the data array
          plotter.plotfft(argsOut[0], argsOut[1], argsOut[2]);

          Console.ReadLine();  // Wait for user to exit application
        }

      catch(Exception exception)
        {
          Console.WriteLine("Error: {0}", exception);
        }
      }

    #endregion
  }
}
```

The program does the following:

- Constructs an input array with values representing a random signal with two sinusoids at 15 and 40 Hz embedded inside of it
- Creates an `MWNumericArray` array that contains the data
- Instantiates a `SignalAnalyzer` object
- Calls the `computefft` method, which computes the FFT, frequency, and the spectral density
- Instantiates a `Plotter` object
- Calls the `plotfft` method, which plots the data
- Uses a `try/catch` block to handle exceptions

The following statement

```
MWNumericArray data= new MWNumericArray(MWArrayComplexity.Real,
MWNumericType.Double, numSamples);
```

shows how to use the `MWArray` class library to construct a `MWNumericArray` that is used as method input to the `computefft` function.

The following statement

```
SignalAnalyzer signalAnalyzer = new SignalAnalyzer();
```

creates an instance of the class `SignalAnalyzer`, and the following statement

```
MWArray[] argsOut= signalAnalyzer.computefft(3, data, interval);
```

calls the method `computefft`.

**6** Build the `SpectraApp` application using Visual Studio .NET.

   **a** The `SpectraCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `SpectraCSApp.csproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **SpectraCSApp.csproj > Open Outside MATLAB**.

   **b** Add a reference to the `MWArray` component, which is *`matlabroot`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`* `\mwarray.dll`. See "Supported Microsoft .NET Framework Versions" on page 3-2 for a list of supported framework versions.

   **c** If necessary, add (or fix the location of) a reference to the `SpectraComp` component which you built in a previous step. (The component, `SpectraComp.dll`, is in the `\SpectraExample\SpectraComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

**7** Build and run the application in Visual Studio .NET.

# Matrix Math Example

**In this section...**

"Purpose" on page 5-20

"Procedure" on page 5-21

"MATLAB Functions to Be Encapsulated" on page 5-26

"Understanding the MatrixMath Program" on page 5-27

## Purpose

The purpose of the example is to show you the following:

- How to assign more than one MATLAB function to a component class
- How to access the component in a C# application (`MatrixMathApp.cs`) by instantiating `Factor` and using the MWArray class library to handle data conversion

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- How to build and run the `MatrixMathApp` application, using the Visual Studio .NET development environment

This example builds a .NET component to perform matrix math. The example creates a program that performs Cholesky, LU, and QR factorizations on a simple tridiagonal matrix (finite difference matrix) with the following form:

```
A = [ 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 ]
```

You supply the size of the matrix on the command line, and the program constructs the matrix and performs the three factorizations. The original matrix and the results are printed to standard output. You may optionally perform the calculations using a sparse matrix by specifying the string "sparse" as the second parameter on the command line.

## Procedure

1. If you have not already done so, copy the files for this example as follows:
   a. Copy the following folder that ships with the MATLAB product to your work folder:

      ```
      matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\MatrixMathExample
      ```

   b. At the MATLAB command prompt, `cd` to the new `MatrixMathExample` subfolder in your work folder.
2. Write the MATLAB functions as you would any MATLAB function.

   The code for the `cholesky`, `ludecomp`, and `qrdecomp` functions is already in your work folder in `MatrixMathExample\MatrixMathComp\`.
3. While in MATLAB, issue the following command to open the Deployment Tool:

   ```
   deploytool
   ```
4. Build the .NET component. See the instructions in “Building Your Component” on page 1-13 for more details. Use the following information:

   | | |
   |---|---|
   | Project Name | `MatrixMathComp` |
   | Class Name | `Factor` |
   | Files to compile | `cholesky.m` `ludecomp.m` `qrdecomp.m` |

5. Write source code for an application that accesses the component.

The sample application for this example is in `MatrixMathExample\MatrixMathCSApp\MatrixMathApp.cs`.

The program listing is shown here.

### MatrixMathApp.cs

```
// *******************************************************************************
//
// MatrixMathApp.css
// This example demonstrates how to use MATLAB Builder NE to build a component
// that returns multiple results and optionally uses sparse matrices for
// arguments.
// Copyright 2001-2010 The MathWorks, Inc.
//
// *******************************************************************************

using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using MatrixMathComp;

namespace MathWorks.Examples.MatrixMath
{
  /// <summary>
  /// This application computes cholesky, LU, and QR factorizations of a finite difference matrix of order N.
  /// The order is passed into the application on the command line.
  /// </summary>
  /// <remarks>
  /// Command Line Arguments:
  /// <newpara></newpara>
  /// args[0] - Matrix order(N)
  /// <newpara></newpara>
  /// args[1] - (optional) sparse; Use a sparse matrix
  /// </remarks>
  class MatrixMathApp
    {
      #region MAIN
```

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
  {
    bool makeSparse= true;
    int matrixOrder= 4;

    MWNumericArray matrix= null; // The matrix to factor

    MWArray argOut= null;     // Stores single factorization result
    MWArray[] argsOut= null;  // Stores multiple factorization results

    try
      {
        // If no argument specified, use defaults
        if (0 != args.Length)
          {
            // Convert matrix order
            matrixOrder= Int32.Parse(args[0]);

            if (0 >= matrixOrder)
              {
                throw new ArgumentOutOfRangeException("matrixOrder", matrixOrder,
                                                      "Must enter a positive integer for the matrix order(N)");
              }

            makeSparse= ((1 < args.Length) && (args[1].Equals("sparse")));
          }

        // Create the test matrix.  If the second argument is "sparse", create a sparse matrix.
     matrix= (makeSparse)
      ? MWNumericArray.MakeSparse(matrixOrder, matrixOrder, MWArrayComplexity.Real, (matrixOrder+(2*(matrixOrder-1))))
      : new MWNumericArray(MWArrayComplexity.Real, MWNumericType.Double, matrixOrder, matrixOrder);

        // Initialize the test matrix
        for (int rowIdx= 1; rowIdx <= matrixOrder; rowIdx++)
          for (int colIdx= 1; colIdx <= matrixOrder; colIdx++)
```

```
          if (rowIdx == colIdx)
            matrix[rowIdx, colIdx]= 2.0;
          else if ((colIdx == rowIdx+1) || (colIdx == rowIdx-1))
            matrix[rowIdx, colIdx]= -1.0;

      // Create a new factor object
      Factor factor= new Factor();

      // Print the test matrix
      Console.WriteLine("Test Matrix:\n{0}\n", matrix);

      // Compute and print the cholesky factorization using the single output syntax
      argOut= factor.cholesky((MWArray)matrix);

      Console.WriteLine("Cholesky Factorization:\n{0}\n", argOut);

      // Compute and print the LU factorization using the multiple output syntax
      argsOut= factor.ludecomp(2, matrix);

      Console.WriteLine("LU Factorization:\nL Matrix:\n{0}\nU Matrix:\n{1}\n", argsOut[0], argsOut[1]);

      MWNumericArray.DisposeArray(argsOut);

      // Compute and print the QR factorization
      argsOut= factor.qrdecomp(2, matrix);

      Console.WriteLine("QR Factorization:\nQ Matrix:\n{0}\nR Matrix:\n{1}\n", argsOut[0], argsOut[1]);

      Console.ReadLine();
    }

  catch(Exception exception)
    {
      Console.WriteLine("Error: {0}", exception);
    }

  finally
    {
      // Free native resources
      if (null != (object)matrix) matrix.Dispose();
```

```
                if (null != (object)argOut) argOut.Dispose();

                MWNumericArray.DisposeArray(argsOut);
              }
          }

        #endregion
      }
    }
```

The statement

```
Factor factor= new Factor();
```

creates an instance of the class `Factor`.

The following statements call the methods that encapsulate the MATLAB functions:

```
argOut= factor.cholesky((MWArray)matrix);
...
argsOut= factor.ludecomp(2, matrix);
...
argsOut= factor.qrdecomp(2, matrix);
...
```

---

**Note** See "Understanding the MatrixMath Program" on page 5-27 for more details about the structure of this program.

---

6. Build the `MatrixMathApp` application using Visual Studio .NET.

   a. The `MatrixMathCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `MatrixMathCSApp.csproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **MatrixMathCSApp.csproj > Open Outside MATLAB**.

   b. Add a reference to the `MWArray` component, which is *`matlabroot`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`*

`\mwarray.dll.` See “Supported Microsoft .NET Framework Versions” on page 3-2 for a list of supported framework versions.

c If necessary, add (or fix the location of) a reference to the `MatrixMathComp` component which you built in a previous step. (The component, `MatrixMathComp.dll`, is in the `\MatrixMathExample\MatrixMathComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

7 Build and run the application in Visual Studio .NET.

## MATLAB Functions to Be Encapsulated

The following code defines the MATLAB functions used in the example.

### cholesky.m

```
function [L] = Cholesky(A)
%CHOLESKY Cholesky factorization of A.
%   L= CHOLESKY(A) returns the Cholesky factorization of A.
%   This file is used as an example for the .NET Builder
%   Language product.

%   Copyright 2001-2010 The MathWorks, Inc.
%   $Revision: 1.1.4.58.6.4 $  $Date: 2010/07/14 16:41:26 $

    L = chol(A);
```

### ludecomp.m

```
function [L,U] = LUDecomp(A)
%LUDECOMP LU factorization of A.
%   [L,U]= LUDECOMP(A) returns the LU factorization of A.
%   This file is used as an example for the .NET Builder
%   Language product.

%   Copyright 2001-2010 The MathWorks, Inc.
%   $Revision: 1.1.4.58.6.4 $  $Date: 2010/07/14 16:41:26 $

    [L,U] = lu(A);
```

### qrdecomp.m

```
function [Q,R] = QRDecomp(A)
%QRDECOMP QR factorization of A.
%   [Q,R]= QRDECOMP(A) returns the QR factorization of A.
%   This file is used as an example for the .NET Builder
%   Language product.

%   Copyright 2001-2010 The MathWorks, Inc.
%   $Revision: 1.1.4.58.6.4 $  $Date: 2010/07/14 16:41:26 $

    [Q,R] = qr(A);
```

## Understanding the MatrixMath Program

The `MatrixMath` program takes one or two arguments from the command line. The first argument is converted to the integer order of the test matrix. If the string `sparse` is passed as the second argument, a sparse matrix is created to contain the test array. The Cholesky, LU, and QR factorizations are then computed and the results are displayed.

The main method has three parts:

- The first part sets up the input matrix, creates a new factor object, and calls the `cholesky`, `ludecomp`, and `qrdecomp` methods. This part is executed inside of a `try` block. This is done so that if an exception occurs during execution, the corresponding `catch` block will be executed.
- The second part is the `catch` block. The code prints a message to standard output to let the user know about the error that has occurred.
- The third part is a `finally` block to manually clean up native resources before exiting.

---

**Note** This optional as the garbage collector will automatically clean-up resources for you.

---

# Phonebook Example

**In this section...**

"Purpose" on page 5-28
"Procedure" on page 5-28

## Purpose

The makephone function takes a structure array as an input, modifies it, and supplies the modified array as an output.

**Note** For complete reference information about the MWArray class hierarchy, see the MWArray class library link on the product roadmap, under "Documentation Set".

## Procedure

1. If you have not already done so, copy the files for this example as follows:

   a. Copy the following folder that ships with MATLAB to your work folder:

      *matlabroot*\dotnetbuilder\Examples\VS8\NET\PhoneBookExample

   b. At the MATLAB command prompt, cd to the new PhoneBookExample subfolder in your work folder.

2. Write the makephone function as you would any MATLAB function.

   The following code defines the makephone function:

```
function book = makephone(friends)
%MAKEPHONE Add a structure to a phonebook structure
%   BOOK = MAKEPHONE(FRIENDS) adds a field to its input structure.
%   The new field EXTERNAL is based on the PHONE field of the original.
%   Copyright 2006-2010 The MathWorks, Inc.

book = friends;
```

```
for i = 1:numel(friends)
    numberStr = num2str(book(i).phone);
    book(i).external = ['(508) 555-' numberStr];
end
```

This code is already in your work folder in `PhoneBookExample\PhoneBookComp\makephone.m`.

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

```
deploytool
```

4 Build the .NET component. See the instructions in “Building Your Component” on page 1-13 for more details. Use the following information:

| | |
|---|---|
| Project Name | `PhoneBookComp` |
| Class Name | `Phonebook` |
| File to compile | `makephone.m` |

5 Write source code for an application that accesses the component.

The sample application for this example is in *`matlabroot`*`\toolbox\dotnetbuilder\Examples\VS8\NET\PhoneBookExample\PhoneBookCSApp\PhoneBookApp.cs`.

The program defines a structure array containing names and phone numbers, modifies it using a MATLAB function, and displays the resulting structure array.

The program listing is shown here.

**PhoneBookApp.cs**

```
// *******************************************************************************
//
// PhoneBookApp.cs
//
// This example demonstrates how to use MATLAB Builder NE to build a simple
```

```
// component that makes use of MATLAB structures as function arguments.
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *******************************************************************************

/* Necessary package imports */
using System;
using System.Collections.Generic;
using System.Text;
using MathWorks.MATLAB.NET.Arrays;
using PhoneBookComp;

namespace MathWorks.Examples.PhoneBookApp
{
  //
  // This class demonstrates the use of the MWStructArray class
  //
  class PhoneBookApp
    {
      static void Main(string[] args)
        {
          PhoneBook thePhonebook = null;  /* Stores deployment class instance */
          MWStructArray friends= null;   /* Sample input data */
          MWArray[] result= null;        /* Stores the result */
          MWStructArray book= null;      /* Ouptut data extracted from result */

          /* Create the new deployment object */
          thePhonebook= new PhoneBook();

          /* Create an MWStructArray with two fields */
          String[] myFieldNames= { "name", "phone" };
          friends= new MWStructArray(2, 2, myFieldNames);

          /* Populate struct with some sample data --- friends and phone number extensions */
          friends["name", 1]= new MWCharArray("Jordan Robert");
          friends["phone", 1]= 3386;
          friends["name", 2]= new MWCharArray("Mary Smith");
          friends["phone", 2]= 3912;
          friends["name", 3]= new MWCharArray("Stacy Flora");
```

```
    friends["phone", 3]= 3238;
    friends["name", 4]= new MWCharArray("Harry Alpert");
    friends["phone", 4]= 3077;

    /* Show some of the sample data */
    Console.WriteLine("Friends: ");
    Console.WriteLine(friends.ToString());

    /* Pass it to an MATLAB function that determines external phone number */
    result= thePhonebook.makephone(1, friends);
    book= (MWStructArray)result[0];

    Console.WriteLine("Result: ");
    Console.WriteLine(book.ToString());

    /* Extract some data from the returned structure */
    Console.WriteLine("Result record 2:");

    Console.WriteLine(book["name", 2]);
    Console.WriteLine(book["phone", 2]);
    Console.WriteLine(book["external", 2]);

    /* Print the entire result structure using the helper function below */
    Console.WriteLine("");
    Console.WriteLine("Entire structure:");

    DispStruct(book);

    Console.ReadLine();
  }

public static void DispStruct(MWStructArray arr)
  {
    Console.WriteLine("Number of Elements: " + arr.NumberOfElements);

    int[] dims= arr.Dimensions;

    Console.Write("Dimensions: " + dims[0]);

    for (int idx= 1; idx < dims.Length; idx++)
```

```
          {
            Console.WriteLine("-by-" + dims[idx]);
          }

        Console.WriteLine("\nNumber of Fields: " + arr.NumberOfFields);
        Console.WriteLine("Standard MATLAB view:");
        Console.WriteLine(arr.ToString());

        Console.WriteLine("Walking structure:");

        string[] fieldNames= arr.FieldNames;

        for (int element= 1; element <= arr.NumberOfElements; element++)
          {
            Console.WriteLine("Element " + element);

            for (int field= 0; field < arr.NumberOfFields; field++)
              {
                MWArray fieldVal= arr[arr.FieldNames[field], element];

                /* Recursively print substructures, give string display of other classes */
                if (fieldVal.GetType() == typeof(MWStructArray))
                  {
                    Console.WriteLine("    " + fieldNames[field] + ": nested structure:");
                    Console.WriteLine("+++ Begin of \"" + fieldNames[field] + "\" nested structure");
                    DispStruct((MWStructArray)fieldVal);
                    Console.WriteLine("+++ End of \"" + fieldNames[field] + "\" nested structure");
                  }

                else
                  {
                    Console.Write("    " + fieldNames[field] + ": ");
                    Console.WriteLine(fieldVal.ToString());
                  }
              }
          }
      }
    }
}
```

The program does the following:

- Creates a structure array, using MWStructArray to represent the example phonebook data.
- Instantiates the `Phonebook` class as `thePhonebook` object, as shown: `thePhonebook = new phonebook();`
- Calls the `makephone` method to create a modified copy of the structure by adding an additional field, as shown: `result = thePhonebook.makephone(1, friends);`

**6** Build the`PhoneBookCSApp` application using Visual Studio .NET.

**a** The `PhoneBookCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PhoneBookCSApp.csproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **PhoneBookCSApp.csproj > Open Outside MATLAB**.

**b** Add a reference to the `MWArray` component, which is *`matlabroot`*`\toolbox\dotnetbuilder\bin\`*`architecture`*`\`*`framework_version`* `\mwarray.dll`.

**c** If necessary, add (or fix the location of) a reference to the `PhoneBookComp` component which you built in a previous step. (The component, `PhoneBookComp.dll`, is in the `\PhoneBookExample\PhoneBookComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

**7** Build and run the application in Visual Studio .NET.

The `PhoneBookApp` program should display the output:

```
Friends:
2x2 struct array with fields:
    name
    phone
Result:
2x2 struct array with fields:
    name
    phone
    external
```

```
Result record 2:
Mary Smith
3912
(508) 555-3912

Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
Walking structure:
Element 1
   name: Jordan Robert
   phone: 3386
   external: (508) 555-3386
Element 2
   name: Mary Smith
   phone: 3912
   external: (508) 555-3912
Element 3
   name: Stacy Flora
   phone: 3238
   external: (508) 555-3238
Element 4
   name: Harry Alpert
   phone: 3077
   external: (508) 555-3077
```

# 6

# Sample Applications (Microsoft Visual Basic .NET)

The sample applications that follow use the same components as those developed in "Deploying a Component Using the Magic Square Example" on page 1-8 and Chapter 5, "Sample Applications (C#)". Instead of C#, the following applications are written in Microsoft Visual Basic .NET. For details about creating the components, see the procedures noted in the beginning of the description for each application. Then follow the steps shown here to use the component in a Visual Basic application.


- "Magic Square Example (Visual Basic)" on page 6-3
- "Create Plot Example (Visual Basic)" on page 6-7
- "Variable Arguments Example (Visual Basic)" on page 6-11
- "Spectral Analysis Example (Visual Basic)" on page 6-15
- "Matrix Math Example (Visual Basic)" on page 6-20
- "Phonebook Example (Visual Basic)" on page 6-25

**Note** The examples for the MATLAB Builder NE product are in `matlabroot\toolbox\dotnetbuilder\Examples\VSversionnumber`, where *matlabroot* is the folder where the MATLAB product is installed and `VSversionnumber` specifies the version of Microsoft Visual Studio .NET you are using (currently `VS8`). If you have Microsoft Visual Studio .NET installed, you can load projects for all the examples by opening the following solution:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSversionnumber\DotNetExamples.sln
```

# Magic Square Example (Visual Basic)

To create the component for this example, see the first several steps in "Deploying a Component Using the Magic Square Example" on page 1-8. After you build the `MagicSquareComp` component, you can build an application that accesses the component as follows.

1 For this example, the application is `MagicSquareApp.vb`.

   You can find `MagicSquareApp.vb` in:

   ```
   matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET
     \MagicSquareExample\MagicSquareVBApp
   ```

   The program listing is as follows.

   **MagicSquareApp.vb**

```
' *******************************************************************************
'
' MagicSquareApp.vb
'
' This example demonstrates how to use MATLAB Builder NE to build a simple
' component returning a magic square and how to convert MWNumericArray types
' to native .NET types.
'
' Copyright 2001-2010 The MathWorks, Inc.
'
' *******************************************************************************

Imports System
Imports System.Reflection
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports MagicSquareComp

Namespace MathWorks.Examples.MagicSquare

    ' <summary>
```

```
    ' The MagicSquareApp class computes a magic square of the user specified size.
    ' </summary>
    ' <remarks>
    ' args[0] - a positive integer representing the array size.
    ' </remarks>
    Class MagicSquareApp

#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)

            Dim arraySize As MWNumericArray = Nothing
            Dim magicSquare As MWNumericArray = Nothing

            Try
                ' Get user specified command line arguments or set default
                If (0 <> args.Length) Then
                    arraySize = New MWNumericArray(Int32.Parse(args(0)), False)
                Else
                    arraySize = New MWNumericArray(4, False)
                End If

                ' Create the magic square object
                Dim magic As MagicSquareClass = New MagicSquareClass

                ' Compute the magic square and print the result
                magicSquare = magic.makesquare(arraySize)

                Console.WriteLine("Magic square of order {0}{1}{2}{3}", arraySize, Chr(10), Chr(10), magicSquare)

                ' Convert the magic square array to a two dimensional native double array
                Dim nativeArray(,) As Double = CType(magicSquare.ToArray(MWArrayComponent.Real), Double(,))

                Console.WriteLine("{0}Magic square as native array:{1}", Chr(10), Chr(10))

                ' Display the array elements:
                Dim index As Integer = arraySize.ToScalarInteger()
```

```
                For i As Integer = 0 To index - 1
                    For j As Integer = 0 To index - 1
                        Console.WriteLine("Element({0},{1})= {2}", i, j, nativeArray(i, j))
                    Next j
                Next i

                Console.ReadLine() 'Wait for user to exit application

            Catch exception As Exception

                Console.WriteLine("Error: {0}", exception)

            End Try
        End Sub
#End Region

    End Class

End Namespace
```

The application you build from this source file does the following:

- Lets you pass a dimension for the magic square from the command line.
- Converts the dimension argument to a MATLAB integer scalar value.
- Declares variables of type `MWNumericArray` to handle data required by the encapsulated `makesquare` function.

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- Creates an instance of the `MagicSquare` class named `magic`.
- Calls the `makesquare` method, which belongs to the `magic` object. The `makesquare` method generates the magic square using the MATLAB `magic` function.

- Displays the array elements on the command line.

**2** Build the application using Visual Studio .NET.

**a** The `MagicSquareVBApp` folder contains a Visual Studio .NET project file for each example. Open the project in Visual Studio .NET for this example by double-clicking `MagicSquareVBApp.vbproj` in Windows Explorer.

**b** If necessary, add a reference to the `MWArray` component, which is *matlabroot*`\dotnetbuilder\bin\`*architecture*`\`*framework_version* `\mwarray.dll.`

**c** If necessary, add a reference to the `MagicSquareComp` component, which is in the `distrib` subfolder.

**d** Build and run the application in Visual Studio.NET.

# Create Plot Example (Visual Basic)

To create the component for this example, see "Simple Plot Example" on page 5-2. Then create a Visual Basic application as follows:

1. Review the sample application for this example in *matlabroot*\toolbox\dotnetbuilder\Examples*VSversionnumber*\NET\PlotExample\PlotVBApp\PlotApp.vb.

   The program listing is shown here.

   **PlotApp.vb**

```
' *******************************************************************************
'
' PlotApp.vb
'
' This example demonstrates how to use MATLAB Builder NE to build a component
' that displays a MATLAB figure window.
'
' Copyright 2001-2010 The MathWorks, Inc.
'
' *******************************************************************************

Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports PlotComp



Namespace MathWorks.Examples.PlotApp

    ' <summary>
    ' This application demonstrates plotting x-y data by graphing a simple
    ' parabola into a MATLAB figure window.
    ' </summary>
    Class PlotDemoApp
```

```
#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)
            Try
                Const numPoints As Integer = 10  ' Number of points to plot
                Dim idx As Integer
                Dim plotValues(,) As Double = New Double(1, numPoints - 1) {}
                Dim coords As MWNumericArray

                'Plot 5x vs x^2
                For idx = 0 To numPoints - 1
                    Dim x As Double = idx + 1
                    plotValues(0, idx) = x * 5
                    plotValues(1, idx) = x * x
                Next idx

                coords = New MWNumericArray(plotValues)

                ' Create a new plotter object
                Dim plotter As Plotter = New Plotter

                ' Plot the values
                plotter.drawgraph(coords)

                Console.ReadLine()  ' Wait for user to exit application

            Catch exception As Exception
                Console.WriteLine("Error: {0}", exception)
            End Try
        End Sub
#End Region
    End Class
End Namespace
```

The program does the following:

- Creates two arrays of double values
- Creates a `Plotter` object
- Calls the `drawgraph` method to plot the equation using the MATLAB `plot` function
- Uses `MWNumericArray` to handle the data needed by the `drawgraph` method to plot the equation

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- Uses a `try-catch` block to catch and handle any exceptions

The statement

```
Dim plotter As Plotter = New Plotter
```

creates an instance of the `Plotter` class, and the statement

```
plotter.drawgraph(coords)
```

calls the method `drawgraph`.

**2** Build the `PlotApp` application using Visual Studio .NET.

**a** The `PlotVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PlotVBApp.vbproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **PlotVBApp.vbproj > Open Outside MATLAB**.

**b** Add a reference to the `MWArray` component, which is `*matlabroot*\toolbox\dotnetbuilder\bin*architecture**framework_version*\mwarray.dll`.

**c** If necessary, add (or fix the location of) a reference to the `PlotComp` component which you built in a

previous step. (The component, `PlotComp.dll`, is in the `\PlotExample\PlotComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

3 Build and run the application in Visual Studio .NET.

# Variable Arguments Example (Visual Basic)

To create the component for this example, see "Passing Variable Arguments" on page 5-7. Then create a Microsoft Visual Basic application as follows:

1 Review the sample application for this example in *matlabroot*`\toolbox\dotnetbuilder\Examples\VS8\NET\VarArgExample\VarArgVBApp\VarArgApp.vb`.

   The program listing is shown here.

   **VarArgApp.vb**

```
' *******************************************************************************
'
'  VarArgApp.vb
'
' This example demonstrates how to use MATLAB Builder NE to build a component
' with a variable number of input and output arguments.
'
' Copyright 2001-2010 The MathWorks, Inc.
'
' *******************************************************************************

Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports VarArgComp


Namespace MathWorks.Demo.VarArgDemoApp


' <summary>
' This application demonstrates how to call components having methods with varargin/vargout arguments.
' </summary>
Class VarArgDemoApp
```

```
#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)

            ' Initialize the input data
            Dim colorSpec As MWNumericArray = New MWNumericArray(New Double() {0.9, 0.0, 0.0})
            Dim data As MWNumericArray = New MWNumericArray(New Integer(,) {{1, 2}, {2, 4}, {3, 6}, {4, 8}, {5, 10}})
            Dim coords() As MWArray = Nothing

            Try

                ' Create a new plotter object
                Dim plotter As Plotter = New Plotter

                'Extract a variable number of two element x and y coordinate vectors from the data array
                coords = plotter.extractcoords(5, data)

                ' Draw a graph using the specified color to connect the variable number of input coordinates.
                ' Return a two column data array containing the input coordinates.
                data = CType(plotter.drawgraph(colorSpec, coords(0), coords(1), coords(2), coords(3), coords(4)), _
                             MWNumericArray)

                Console.WriteLine("result={0}{1}", Chr(10), data)

                Console.ReadLine()  ' Wait for user to exit application

                ' Note: You can also pass in the coordinate array directly.
                data = CType(plotter.drawgraph(colorSpec, coords), MWNumericArray)

                Console.WriteLine("result=\{0}{1}", Chr(10), data)

                Console.ReadLine()  ' Wait for user to exit application


            Catch exception As Exception
                Console.WriteLine("Error: {0}", exception)
            End Try
```

```
        End Sub
#End Region

    End Class
End Namespace
```

The program does the following:

- Initializes three arrays (`colorSpec`, `data`, and `coords`) using the `MWArray` class library
- Creates a `Plotter` object
- Calls the `extracoords` and `drawgraph` methods
- Uses `MWNumericArray` to handle the data needed by the methods

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- Uses a `try-catch-finally` block to catch and handle any exceptions

The following statements are alternative ways to call the `drawgraph` method:

```
data = CType(plotter.drawgraph(colorSpec, coords(0), coords(1), coords(2), coords(3), coords(4)), MWNumericArray)
...
data = CType(plotter.drawgraph(colorSpec, coords), MWNumericArray)
```

**2** Build the `VarArgApp` application using Visual Studio .NET.

**a** The `VarArgVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `VarArgVBApp.vbproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **VarArgVBApp.vbproj > Open Outside MATLAB**.

b. Add a reference to the `MWArray` component, which is *matlabroot*`\toolbox\dotnetbuilder\bin\`*architecture*`\`*framework_version*`\mwarray.dll`.

c. If necessary, add (or update the location of) a reference to the `VarArgComp` component which you built in a previous step. (The component, `VarArgComp.dll`, is in the `\VarArgExample\VarArgComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

3. Build and run the application in Visual Studio .NET.

# Spectral Analysis Example (Visual Basic)

To create the component for this example, see the first few steps of the "Spectral Analysis Example" on page 5-13. Then create a Microsoft Visual Basic application as follows:

**1** Review the sample application for this example in `matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\SpectraVBApp\SpectraApp.vb`.

The program listing is shown here.

### SpectraApp.vb

```
' *******************************************************************************
'
'SpectraApp.vb
'
' This example demonstrates how to use MATLAB Builder NE to build a component
' with multiple classes.
'
' Copyright 2001-2010 The MathWorks, Inc.
'
' *******************************************************************************

Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports SpectraComp


Namespace MathWorks.Examples.SpectraApp

    ' <summary>
    ' This application computes and plots the power spectral density of an input signal.
    ' </summary>
    Class SpectraDemoApp
```

```
#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)
            Try
                Const interval As Double = 0.01   ' The sampling interval
                Const numSamples As Integer = 1001   ' The number of samples

                ' Construct input data as sin(2*PI*15*t) + (sin(2*PI*40*t) plus a
                ' random signal.  Duration= 10; Sampling interval= 0.01
                Dim data As MWNumericArray = New MWNumericArray(MWArrayComplexity.Real,
                                             MWNumericType.Double, numSamples)

                Dim random As Random = New Random

                ' Initialize data
                Dim t As Double
                Dim idx As Integer
                For idx = 1 To numSamples
                    t = (idx - 1) * interval
                    data(idx) = New MWNumericArray(Math.Sin(2.0 * Math.PI * 15.0 * t) +
                                    Math.Sin(2.0 * Math.PI * 40.0 * t) + random.NextDouble())
                Next idx

                ' Create a new signal analyzer object
                Dim signalAnalyzer As SignalAnalyzer = New SignalAnalyzer

                ' Compute the fft and power spectral density for the data array
                Dim argsOut() As MWArray = signalAnalyzer.computefft(3, data,
                                                 MWArray.op_Implicit(interval))

                ' Print the first twenty elements of each result array
                Dim numElements As Integer = 20
                Dim resultArray As MWNumericArray =
                         New MWNumericArray(MWArrayComplexity.Complex,
                                    MWNumericType.Double, numElements)

                For idx = 1 To numElements
```

```
                resultArray(idx) = (CType(argsOut(0), MWNumericArray))(idx)
            Next idx

            Console.WriteLine("FFT:{0}{1}{2}", Chr(10), resultArray, Chr(10))

            For idx = 1 To numElements
                resultArray(idx) = (CType(argsOut(1), MWNumericArray))(idx)
            Next idx

            Console.WriteLine("Frequency:{0}{1}{2}", Chr(10), resultArray, Chr(10))

            For idx = 1 To numElements
                resultArray(idx) = (CType(argsOut(2), MWNumericArray))(idx)
            Next idx

            Console.WriteLine("Power Spectral Density:{0}{1}{2}", Chr(10), resultArray, Chr(10))

            ' Create a new plotter object
            Dim plotter As Plotter = New Plotter

            ' Plot the fft and power spectral density for the data array
            plotter.plotfft(argsOut(0), argsOut(1), argsOut(2))

            Console.ReadLine()  ' Wait for user to exit application

        Catch exception As Exception
            Console.WriteLine("Error: {0}", exception)

        End Try
    End Sub
#End Region

  End Class
End Namespace
```

The program does the following:

- Constructs an input array with values representing a random signal with two sinusoids at 15 and 40 Hz embedded inside of it

- Uses `MWNumericArray` to handle data conversion

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

- Instantiates a `SignalAnalyzer` object
- Calls the `computefft` method, which computes the FFT, frequency, and the spectral density
- Instantiates a `Plotter` object
- Calls the `plotfft` method, which plots the data
- Uses a `try/catch` block to handle exceptions

The following statements

```
Dim data As MWNumericArray = New MWNumericArray_
    (MWArrayComplexity.Real, MWNumericType.Double, numSamples)
...
Dim resultArray As MWNumericArray = New MWNumericArray_
    (MWArrayComplexity.Complex, MWNumericType.Double, numElements)
```

show how to use the MWArray class library to construct the necessary data types.

The following statement

```
Dim signalAnalyzer As SignalAnalyzer = New SignalAnalyzer
```

creates an instance of the class `SignalAnalyzer`, and the following statement

```
Dim argsOut() As MWArray = signalAnalyzer.computefft
               (3, data, MWArray.op_Implicit(interval))
```

calls the method `computefft` and request three outputs.

2 Build the `SpectraApp` application using Visual Studio .NET.

a. The `SpectraVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `SpectraVBApp.vbproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **SpectraVBApp.vbproj > Open Outside MATLAB**.

b. Add a reference to the `MWArray` component, which is *matlabroot*`\toolbox\dotnetbuilder\bin\`*architecture*`\`*framework_version* `\mwarray.dll`.

c. If necessary, add (or update the location of) a reference to the `SpectraComp` component which you built in a previous step. (The component, `SpectraComp.dll`, is in the `\SpectraExample\SpectraComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

3. Build and run the application in Visual Studio .NET.

# Matrix Math Example (Visual Basic)

To create the component for this example, see the first few steps in "Matrix Math Example" on page 5-20. Then create a Microsoft Visual Basic application as follows.

1 Review the sample application for this example in:

   *matlabroot*`\toolbox\dotnetbuilder\Examples\VS8\NET\MatrixMathExample`
   `\MatrixMathVBApp\MatrixMathApp.vb.`

   The program listing is shown here.

   **MatrixMathApp.vb**

```
' *******************************************************************************
'
' MatrixMathApp.vb
'
' This example demonstrates how to use MATLAB Builder NE to build a component
' that returns multiple results and optionally uses sparse matrices for
' arguments.
' Copyright 2001-2010 The MathWorks, Inc.
'
' *******************************************************************************

Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports MatrixMathComp


Namespace MathWorks.Demo.MatrixMathApp

    ' <summary>
    ' This application computes cholesky, LU, and QR factorizations of a
    '  finite difference matrix of order N.
    ' The order is passed into the application on the command line.
```

```
    ' </summary>
    ' <remarks>
    ' Command Line Arguments:
    ' <newpara></newpara>
    ' args[0] - Matrix order(N)
    ' <newpara></newpara>
    ' args[1] - (optional) sparse; Use a sparse matrix
    ' </remarks>
    Class MatrixMathDemoApp

#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)

            Dim makeSparse As Boolean = True
            Dim matrixOrder As Integer = 4

            Dim matrix As MWNumericArray = Nothing ' The matrix to factor

            Dim argOut As MWArray = Nothing   ' Stores single factorization result
            Dim argsOut() As MWArray = Nothing  ' Stores multiple factorization results

            Try
                ' If no argument specified, use defaults
                If (0 <> args.Length) Then
                    'Convert matrix order
                    matrixOrder = Int32.Parse(args(0))

                    If (0 > matrixOrder) Then
                        Throw New ArgumentOutOfRangeException("matrixOrder", matrixOrder, _
                                                              "Must enter a positive integer for the matrix order(N)")
                    End If

                    makeSparse = ((1 < args.Length) AndAlso (args(1).Equals("sparse")))
                End If

                ' Create the test matrix.  If the second argument is "sparse", create a sparse matrix.
```

```
matrix = IIf(makeSparse, _
MWNumericArray.MakeSparse(matrixOrder, matrixOrder, MWArrayComplexity.Real,
                                 (matrixOrder + (2 * (matrixOrder - 1)))), _
New MWNumericArray(MWArrayComplexity.Real, MWNumericType.Double, matrixOrder, matrixOrder))

' Initialize the test matrix
For rowIdx As Integer = 1 To matrixOrder
    For colIdx As Integer = 1 To matrixOrder
        If rowIdx = colIdx Then
            matrix(rowIdx, colIdx) = New MWNumericArray(2.0)
        ElseIf colIdx = rowIdx + 1 Or colIdx = rowIdx - 1 Then
            matrix(rowIdx, colIdx) = New MWNumericArray(-1.0)
        End If
    Next colIdx
Next rowIdx

' Create a new factor object
Dim factor As Factor = New Factor

' Print the test matrix
Console.WriteLine("Test Matrix:{0}{1}{2}", Chr(10), matrix, Chr(10))

' Compute and print the cholesky factorization using the single output syntax
argOut = factor.cholesky(matrix)

Console.WriteLine("Cholesky Factorization:{0}{1}{2}", Chr(10), argOut, Chr(10))

' Compute and print the LU factorization using the multiple output syntax
argsOut = factor.ludecomp(2, matrix)

Console.WriteLine("LU Factorization:{0}L Matrix:{1}{2}{3}U Matrix:{4}{5}{6}", Chr(10), Chr(10),
                            argsOut(0), Chr(10), Chr(10), argsOut(1), Chr(10))

MWNumericArray.DisposeArray(argsOut)

' Compute and print the QR factorization
argsOut = factor.qrdecomp(2, matrix)

Console.WriteLine("QR Factorization:{0}Q Matrix:{1}{2}{3}R Matrix:{4}{5}{6}", Chr(10),
                      Chr(10), argsOut(0), Chr(10), Chr(10), argsOut(1), Chr(10))
```

```
                Console.ReadLine()

            Catch exception As Exception

                Console.WriteLine("Error: {0}", exception)

            Finally

                ' Free native resources
                If Not (matrix Is Nothing) Then
                    matrix.Dispose()
                End If
                If Not (argOut Is Nothing) Then
                    argOut.Dispose()
                End If

                MWNumericArray.DisposeArray(argsOut)
            End Try
        End Sub
#End Region
    End Class
End Namespace
```

The statement

```
Dim factor As Factor = New Factor
```

creates an instance of the class `Factor`.

The following statements call the methods that encapsulate the MATLAB functions:

```
argOut = factor.cholesky(matrix)


argsOut = factor.ludecomp(2, matrix)

...
argsOut = factor.qrdecomp(2, matrix)
```

---

**Note** See "Understanding the MatrixMath Program" on page 5-27 for more details about the structure of this program.

---

2. Build the `MatrixMathApp` application using Visual Studio .NET.

   a. The `MatrixMathVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `MatrixMathVBApp.vbproj` in Windows Explorer. You can also open it from the MATLAB desktop by right-clicking **MatrixMathVBApp.vbproj > Open Outside MATLAB**.

   b. Add a reference to the `MWArray` component, which is `*matlabroot*\toolbox\dotnetbuilder\bin*architecture**framework_version*\mwarray.dll`.

   c. If necessary, add (or update the location of) a reference to the `MatrixMathComp` component which you built in a previous step. (The component, `MatrixMathComp.dll`, is in the `\MatrixMathExample\MatrixMathComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

3. Build and run the application in Visual Studio .NET.

# Phonebook Example (Visual Basic)

**In this section...**

"makephone Function" on page 6-25

"Procedure" on page 6-25

## makephone Function

The `makephone` function takes a structure array as an input, modifies it, and supplies the modified array as an output.

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray class library link on the product roadmap, under "Documentation Set".

---

## Procedure

1. If you have not already done so, copy the files for this example as follows:

   a. Copy the following folder that ships with MATLAB to your work folder:

      *matlabroot*`\dotnetbuilder\Examples\VS8\NET\PhoneBookExample`

   b. At the MATLAB command prompt, `cd` to the new `PhoneBookExample` subfolder in your work folder.

2. Write the `makephone` function as you would any MATLAB function.

   The following code defines the `makephone` function:

```
function book = makephone(friends)
%MAKEPHONE Add a structure to a phonebook structure
%   BOOK = MAKEPHONE(FRIENDS) adds a field to its input structure.
%   The new field EXTERNAL is based on the PHONE field of the original.
%   This file is used as an example for MATLAB
%   Builder for Java.
```

```
%   Copyright 2006-2010 The MathWorks, Inc.

book = friends;
for i = 1:numel(friends)
    numberStr = num2str(book(i).phone);
    book(i).external = ['(508) 555-' numberStr];
end
```

This code is already in your work folder in `PhoneBookExample\PhoneBookComp\makephone.m`.

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

```
deploytool
```

4 Build the .NET component. See the instructions in "Building Your Component" on page 1-13 for more details. Use the following information:

| | |
|---|---|
| Project Name | `PhoneBookComp` |
| Class Name | `phonebook` |
| File to compile | `makephone.m` |

5 Write source code for an application that accesses the component.

The sample application for this example is in *`matlabroot`*`\toolbox\dotnetbuilder\Examples\VS8\NET PhoneBookExample\PhoneBookVBApp\PhoneBookApp.vb`.

The program defines a structure array containing names and phone numbers, modifies it using a MATLAB function, and displays the resulting structure array.

The program listing is shown here.

**PhoneBookApp.vb**

```
' ******************************************************************************
'
```

```
' PhoneBookApp.vb
'
' This example demonstrates how to use MATLAB Builder NE to build a simple
' component that makes use of MATLAB structures as function arguments.
'
' Copyright 2001-2010 The MathWorks, Inc.
'
' ******************************************************************************

' Necessary package imports

Imports MathWorks.MATLAB.NET.Arrays
Imports PhoneBookComp

'
' getphone class demonstrates the use of the MWStructArray class
'
Public Module PhoneBookVBApp
    Public Sub Main()
        Dim thePhonebook As phonebook   'Stores deployment class instance
        Dim friends As MWStructArray    'Sample input data
        Dim result As Object()          'Stores the result
        Dim book As MWStructArray       'Ouptut data extracted from result

        ' Create the new deployment object
        thePhonebook = New phonebook()

        ' Create an MWStructArray with two fields
        Dim myFieldNames As String() = {"name", "phone"}
        friends = New MWStructArray(2, 2, myFieldNames)

        ' Populate struct with some sample data --- friends and phone numbers
        friends("name", 1) = New MWCharArray("Jordan Robert")
        friends("phone", 1) = 3386
        friends("name", 2) = New MWCharArray("Mary Smith")
        friends("phone", 2) = 3912
        friends("name", 3) = New MWCharArray("Stacy Flora")
        friends("phone", 3) = 3238
        friends("name", 4) = New MWCharArray("Harry Alpert")
        friends("phone", 4) = 3077
```

```
        ' Show some of the sample data
        Console.WriteLine("Friends: ")
        Console.WriteLine(friends.ToString())

        ' Pass it to an MATLAB function that determines external phone number
        result = thePhonebook.makephone(1, friends)
        book = CType(result(0), MWStructArray)
        Console.WriteLine("Result: ")
        Console.WriteLine(book.ToString())

        ' Extract some data from the returned structure '
        Console.WriteLine("Result record 2:")

        Console.WriteLine(book("name", 2))
        Console.WriteLine(book("phone", 2))
        Console.WriteLine(book("external", 2))

        ' Print the entire result structure using the helper function below
        Console.WriteLine("")
        Console.WriteLine("Entire structure:")
        dispStruct(book)
    End Sub


    Sub dispStruct(ByVal arr As MWStructArray)
        Console.WriteLine("Number of Elements: " + arr.NumberOfElements.ToString())
        'int numDims = arr.NumberofDimensions
        Dim dims As Integer() = arr.Dimensions
        Console.Write("Dimensions: " + dims(0).ToString())

        Dim i As Integer
        For i = 1 To dims.Length
            Console.WriteLine("-by-" + dims(i - 1).ToString())
        Next i
        Console.WriteLine("")
        Console.WriteLine("Number of Fields: " + arr.NumberOfFields.ToString())
        Console.WriteLine("Standard MATLAB view:")
        Console.WriteLine(arr.ToString())
        Console.WriteLine("Walking structure:")
```

```
        Dim fieldNames As String() = arr.FieldNames

        Dim element As Integer
        For element = 1 To arr.NumberOfElements
            Console.WriteLine("Element " + element.ToString())
            Dim field As Integer
            For field = 0 To arr.NumberOfFields - 1
                Dim fieldVal As MWArray = arr(arr.FieldNames(field), element)
                ' Recursively print substructures, give string display of other classes
                If (TypeOf fieldVal Is MWStructArray) Then
                    Console.WriteLine("   " + fieldNames(field) + ": nested structure:")
                    Console.WriteLine("+++ Begin of \"" + fieldNames[field] + " \ " nested structure")

                    dispStruct(CType(fieldVal, MWStructArray))
                    Console.WriteLine("+++ End of \"" + fieldNames[field] + " \ " nested structure")
                Else
                    Console.Write("   " + fieldNames(field) + ": ")
                    Console.WriteLine(fieldVal.ToString())
                End If
            Next field
        Next element
    End Sub
End Module
```

The program does the following:

- Creates a structure array, using MWStructArray to represent the example phonebook data.
- Instantiates the plotter class as `thePhonebook` object, as shown: `thePhonebook = new phonebook();`
- Calls the `makephone` method to create a modified copy of the structure by adding an additional field, as shown: `result = thePhonebook.makephone(1, friends);`

**6** Build the`PhoneBookVBApp` application using Visual Studio .NET.

**a** The `PhoneBookVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PhoneBookVBApp.vbproj` in Windows Explorer.

You can also open it from the MATLAB desktop by right-clicking **PhoneBookVBApp.vbproj > Open Outside MATLAB**.

b Add a reference to the `MWArray` component, which is `*matlabroot*\toolbox\dotnetbuilder\bin*architecture**framework_version*\mwarray.dll`.

c If necessary, add (or fix the location of) a reference to the `PhoneBookVBComp` component which you built in a previous step. (The component, `PhoneBookComp.dll`, is in the `\PhoneBookExample\PhoneBookVBApp\x86\V2.0\Debug\distrib` subfolder of your work area.)

7 Build and run the application in Visual Studio .NET.

The `getphone` program should display the output:

```
Friends:
2x2 struct array with fields:
    name
    phone
Result:
2x2 struct array with fields:
    name
    phone
    external
Result record 2:
Mary Smith
3912
(508) 555-3912

Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
Walking structure:
```

```
Element 1
   name: Jordan Robert
   phone: 3386
   external: (508) 555-3386
Element 2
   name: Mary Smith
   phone: 3912
   external: (508) 555-3912
Element 3
   name: Stacy Flora
   phone: 3238
   external: (508) 555-3238
Element 4
   name: Harry Alpert
   phone: 3077
   external: (508) 555-3077
```

# Deploying a MATLAB Figure Over the Web Using WebFigures


- "About the WebFigures Feature" on page 7-2
- "Before You Use WebFigures" on page 7-3
- "Quick Start: Implementing a WebFigure" on page 7-7
- "Advanced Configuration of a WebFigure" on page 7-15
- "Upgrading Your WebFigures" on page 7-30
- "Troubleshooting" on page 7-31
- "Logging Levels" on page 7-33

# About the WebFigures Feature

Using the WebFigures feature in MATLAB Builder NE you can 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.

This chapter includes "Quick Start: Implementing a WebFigure" on page 7-7, which guides you through implementing the basic features of WebFigures, and an advanced section to let you customize your configuration depending on differing server architectures.

## Supported Renderers for WebFigures

The MATLAB Builder NE WebFigures feature uses the same renderer used when the figure was originally created by default.

In MATLAB, the renderer is either explicitly specified for a figure or determined by the data being plotted. For more information about supported renderers in MATLAB, see http://www.mathworks.com/support/tech-notes/1200/1201.html.

---

**Note** The WebFigures feature does not support the `Painter` renderer due to technical limitations. If this renderer is requested, the renderer `Zbuffer` will be invoked before the data is displayed on the Web page.

---

# Before You Use WebFigures

**In this section...**

"Your Role in the .NET WebFigure Deployment Process" on page 7-3

"What You Need to Know to Implement WebFigures" on page 7-5

"Required Products" on page 7-5

"Assumptions About the Examples" on page 7-6

## Your Role in the .NET WebFigure Deployment Process

Depending on your role in your organization, as well as a number of other criteria, you may need to implement either the beginning or the advanced configuration of WebFigures.

The table WebFigures for .NET Deployment Roles, Responsibilities, and Tasks on page 7-3 describes some of the different roles, or jobs, that MATLAB Builder NE users typically perform and which method of configuration they would most likely use when running "Quick Start: Implementing a WebFigure" on page 7-7 and "Advanced Configuration of a WebFigure" on page 7-15.

**WebFigures for .NET Deployment Roles, Responsibilities, and Tasks**

| Role | Typical Responsibilities | Tasks |
| --- | --- | --- |
| MATLAB programmer | • Understand end-user business requirements and the mathematical models needed to support them.<br>• Write MATLAB code.<br>• Build an executable component with MATLAB tools (usually with support from a .NET programmer).<br>• Package the component for distribution to end users. | • Write and deploy MATLAB code, such as that in "Assumptions About the Examples" on page 7-6. |

**WebFigures for .NET Deployment Roles, Responsibilities, and Tasks (Continued)**

| Role | Typical Responsibilities | Tasks |
|---|---|---|
| .NET programmer (business-service developer or front-end developer) | • Design and configure the IT environment, architecture, or infrastructure.<br>• Install deployable applications along with the proper version of the MCR.<br>• Create mechanisms for exposing application functionality to the end user. | • Uses "Quick Start: Implementing a WebFigure" on page 7-7 to easily create a graphic, such as a MATLAB figure, that the end user can manipulate over the Web.<br>• Use the "Advanced Configuration of a WebFigure" on page 7-15 to create a flexible, scalable implementation that can meet a number of varied architectural requirements. |

## What You Need to Know to Implement WebFigures

The following knowledge is assumed when you implement WebFigures for .NET:

- If you are a MATLAB programmer:
  - A basic knowledge of MATLAB
- If you are a .NET programmer:
  - Knowledge of how to build a Web site using Microsoft Visual Studio.
  - Experience deploying MATLAB applications

## Required Products

Install the following products to implement WebFigures for .NET, depending on your role.

| MATLAB Programmer | .NET Programmer |
|---|---|
| MATLAB R2008b or later | Microsoft Visual Studio 2005 or later |
| MATLAB Compiler | Microsoft .NET Framework 2.0 or later |
| MATLAB Builder NE | MATLAB Compiler Runtime version 7.9 or later |

## Assumptions About the Examples

To work with the examples in this chapter:

- Assume the following MATLAB function has been created:

```
function df = getKnot()
        f = figure('Visible','off');  %Create a figure.
                                     %Make sure it isn't visible.
        knot;                        %Put something into the figure.
        df = webfigure(f);           %Give the figure to your function
                                     % and return the result.
        close(f);                    %Close the figure.
end
```

- Assume that the function `getKnot` has been deployed in a .NET component (using Chapter 1, "Getting Started" for example) with a namespace of `MyComponent.MyComponentclass`.
- Assume the MATLAB Compiler Runtime (MCR) has been installed. If not, refer to "Install Your Application on Target Computers Without MATLAB Using the MATLAB Compiler Runtime (MCR)" in the MATLAB Compiler documentation.
- If you are running on a system with 64-bit architecture, use the information in "Advanced Configuration of a WebFigure" on page 7-15 to work with WebFigures unless you are deploying a Web site which is 32-bit only and you have a 32-bit MCR installed.

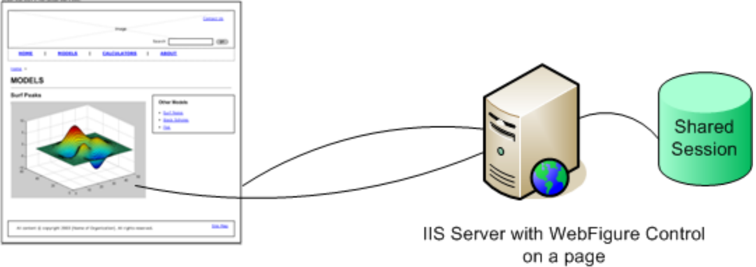
# Quick Start: Implementing a WebFigure

| In this section... |
|---|
| "Overview" on page 7-7 |
| "Procedure" on page 7-7 |

## Overview

Using Quick Start, both the WebFigure service and the page that has the WebFigure embedded on it reside on a single server. This configuration enables you to quickly drag and drop the `WebFigureControl` on a Web page.

**Using the WebFigure Control on the Frontend Servers Outside the Firewall**




## Procedure

To implement WebFigures for MATLAB Builder NE using the Quick Start approach, do the following. For more information about the Quick Start option, see "About the WebFigures Feature" on page 7-2.

1. Start Microsoft Visual Studio.

2. Select **File > New > Web Site** to open.

3. Select one of the template options and click **OK**.

---

**Caution** Do not select **Empty Web Site** as it is not possible to create a WebFigure using this option.

---

4 Add `WebFigureControl` to the Microsoft Visual Studio toolbar by dragging the file *`InstallRoot`*`\toolbox\dotnetbuilder\bin\`*`arch`*`\v2.0\WebFiguresService.dll`, (where *`InstallRoot`* is the location of the installed MCR for machines with an installed MCR and *`matlabroot`* on a MATLAB Builder NE development machine without the MCR installed), on to the Microsoft Visual Studio **Toolbox** toolbar as follows:

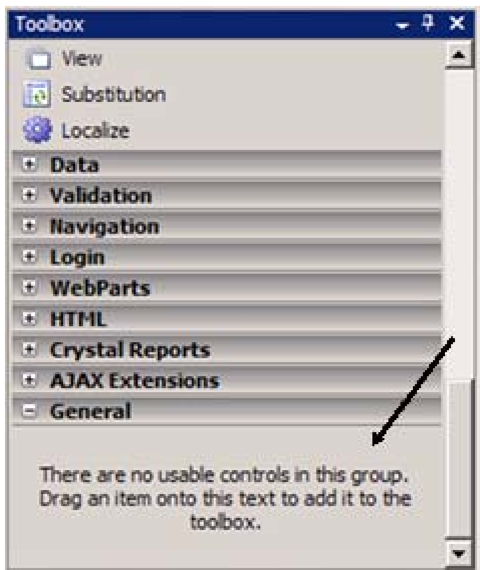
---

**Note** If you are running on a system with 64-bit architecture, use the information in "Advanced Configuration of a WebFigure" on page 7-15 to work with WebFigures unless you are deploying a Web site which is 32-bit only and you have a 32-bit MCR installed.

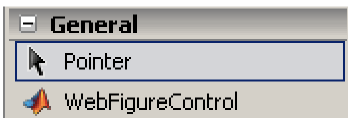
---

a Expand the **General** section of the **Toolbox** toolbar.

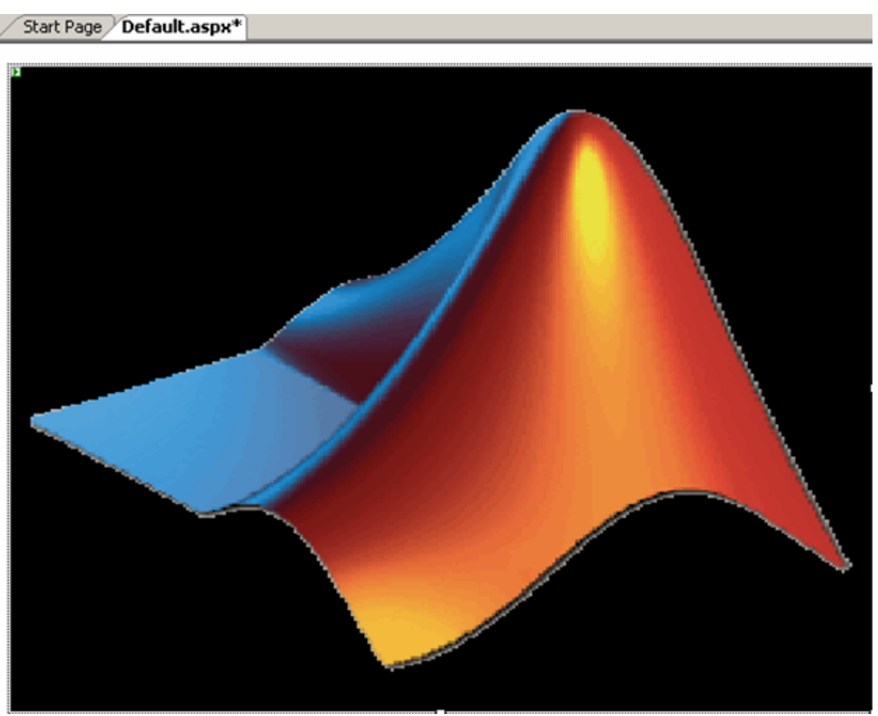
b Using your mouse, drag the DLL file to the expanded section, as shown by the arrow:




If you added the control correctly, you will see the following `WebFigureControl` in the **General** section of the Microsoft Visual Studio toolbar:



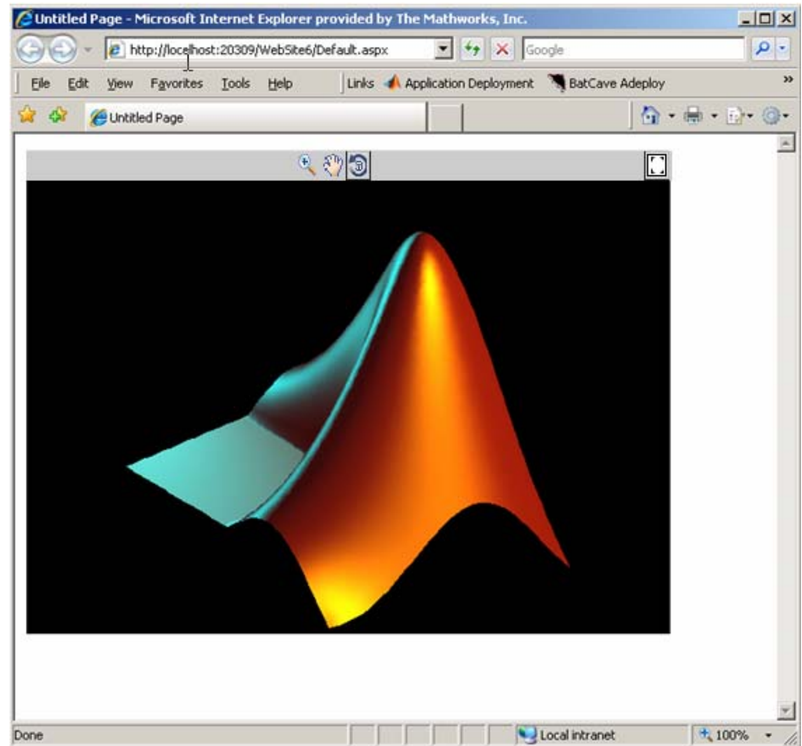

**5** Drag the `WebFigureControl` from the toolbar to your Web page. After dragging, the Web page displays the following default figure.




You can resize the control as you would any other .NET Web control.

6. Switch to the Design view in Microsoft Visual Studio by selecting **View > Designer**.

7. Test the Web page by “playing” it in Microsoft Visual Studio. Select **Debug > Start Debugging**. The page should appear as follows.




**8** Interact with the default figure on the page using your mouse. Click one of the three control icons at the top of the figure to activate the desired control, select the desired region of the figure you want to manipulate, then click and drag as appropriate. For example, to zoom in on the figure, click the magnifying glass icon, then hover over the figure.

**9** Close the page as you would any other window, automatically exiting debug or "play" mode.

**10** The `WebFigureService` you created has been verified as functioning properly and you can attach a custom WebFigure to the Web page:

**a** To enable return of the `webfigure` and to bind it to the `webfigure` control, add a reference to `MWArray` to your project and a reference to the deployed component you created earlier (in "Assumptions About

the Examples" on page 7-6). See Chapter 4, "Integrating Your .NET Component" for more information.

**b** In Microsoft Visual Studio, access the code for the Web page by selecting **View > Code**.

**c** In Microsoft Visual Studio, go to the `Page_Load` method, and add this code, depending on if you are using the C# or Visual Basic language. Adding code to the `Page_Load` method ensures it executes every time the Web page loads.

---

**Note** The following code snippets belong to the partial classes generated by your .NET Web page.

---

- **C#:**

```
using MyComponent;
using MathWorks.MATLAB.NET.WebFigures;

protected void Page_Load(object sender, EventArgs e)
{
        MyComponentclass myDeployedComponent =
                new MyComponentclass();
        WebFigureControl1.WebFigure =
                new WebFigure(myDeployedComponent.getKnot());
}
```

- **Visual Basic:**

```
Imports MyComponent
Imports MathWorks.MATLAB.NET.WebFigures

Protected Sub Page_Load(ByVal sender As Object,
                        ByVal e As System.EventArgs)
                        Handles Me.Load
    Dim myDeployedComponent As _
         New MyComponentclass()
    WebFigureControl1.WebFigure = _
         New WebFigure(myDeployedComponent.getKnot())
```

```
End Sub
```

---

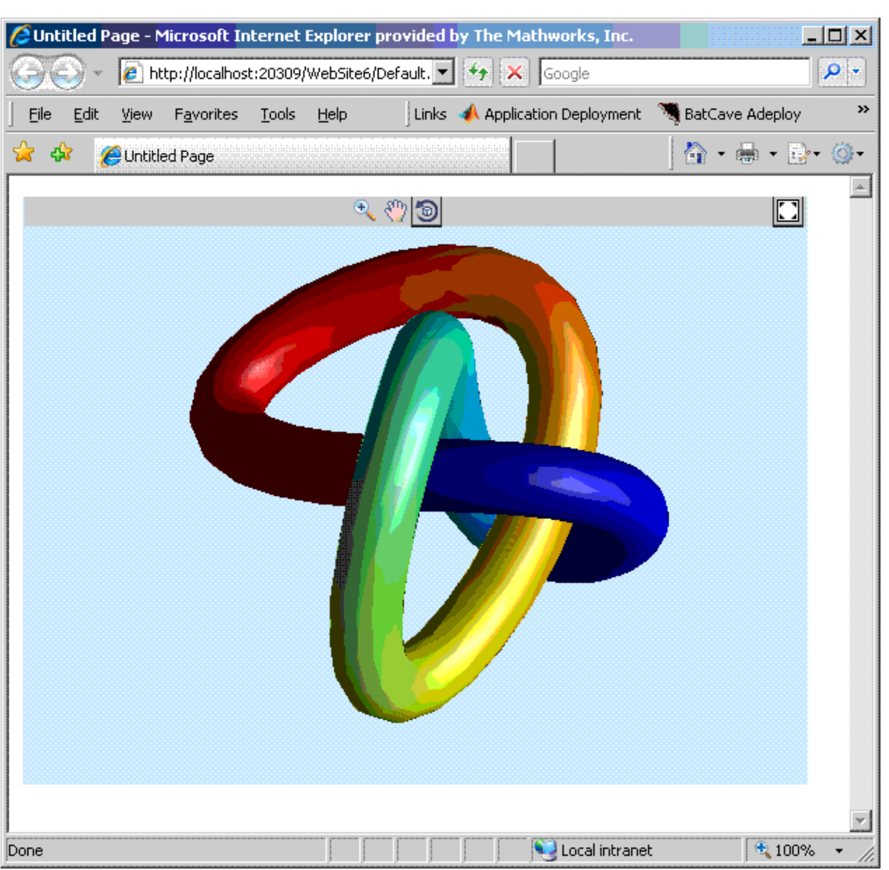
**Tip** This code causes the deployed component to be reinitialized upon each refresh of the page. A better implementation would involve initializing the `myDeployedComponent` variable when the server starts up using a `Global.asax` file, and then using that variable to get the `WebFigure` object. For more information on `Global.asax`, see "Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up" on page 7-28.

---

**Note** `WebFigureControl` stores the `WebFigure` object in the IIS session cache for each individual user. If this is not the desired configuration, see "Advanced Configuration of a WebFigure" on page 7-15 for information on creating a custom configuration.

---

11 Replay the Web page in Microsoft Visual Studio to confirm your WebFigure appears as desired. It should look like this.


Untitled Page - Microsoft Internet Explorer provided by The Mathworks, Inc.
http://localhost:20309/WebSite6/Default.
Google
File Edit View Favorites Tools Help
Links Application Deployment BatCave Adeploy
Untitled Page
Done
Local intranet
100%

# Advanced Configuration of a WebFigure

**In this section...**

"Overview" on page 7-15

"Manually Installing WebFigureService" on page 7-17

"Retrieving Multiple WebFigures From a Component" on page 7-18

"Attaching a WebFigure" on page 7-21

"Setting Up WebFigureControl for Remote Invocation" on page 7-23

"Getting an Embeddable String That References a WebFigure Attached to a WebFigureService" on page 7-25

"Improving Processing Times for JavaScript Using Minification" on page 7-27

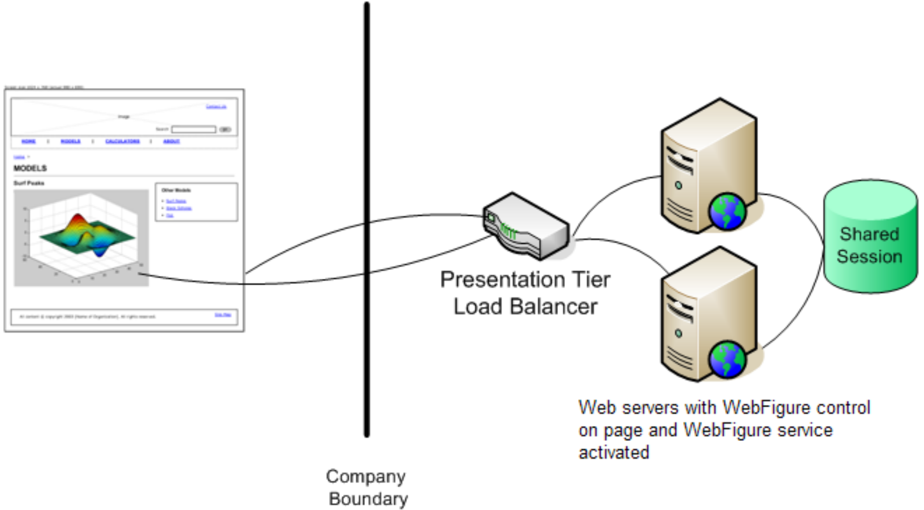
"Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up" on page 7-28

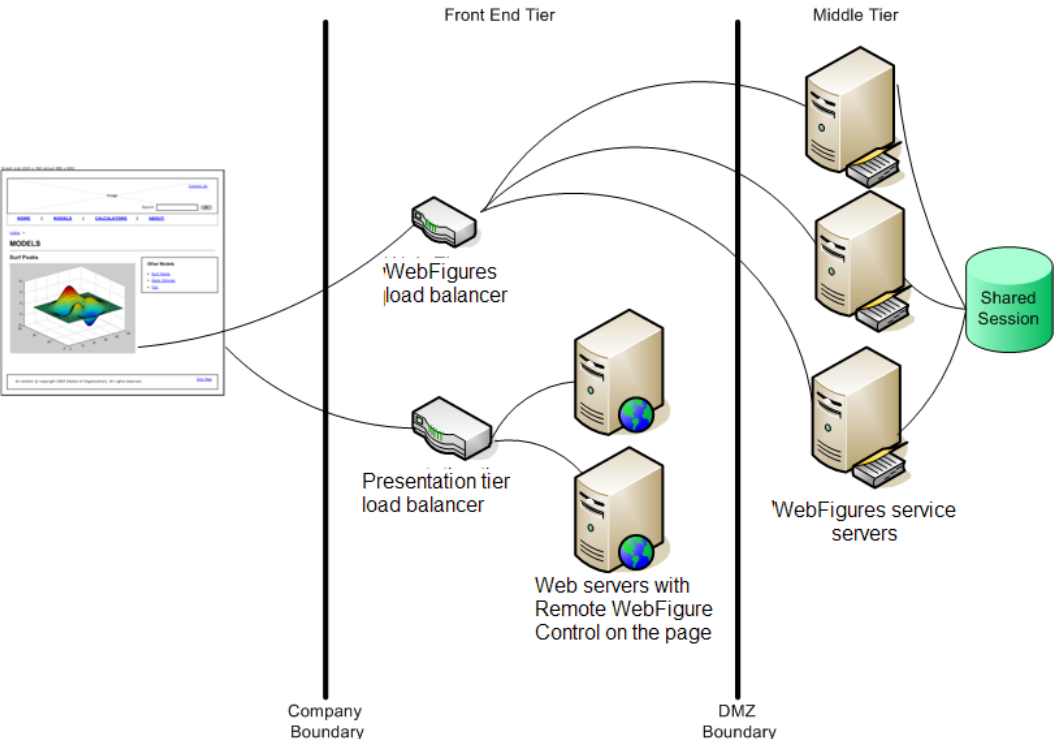
## Overview

The advanced configuration gives the experienced .NET programmer (possibly a business service developer or front-end developer) flexibility and control in configuring system architecture based on differing needs. For example, with the `WebFigureService` and the Web page on different servers, the administrator can optimally position the MCR (for performance reasons) or place customer-sensitive customer data behind a security firewall, if needed.

In summary, the advanced configuration offers more choices and adaptability for the user more familiar with Web environments and related technology, as illustrated by the following graphics.

This section describes various ways to customize the basic WebFigures implementation described in "Quick Start: Implementing a WebFigure" on page 7-7.


Using the WebFigure Control on the
Frontend Servers Outside the Firewall
MODELS
Surf Peaks
Presentation Tier
Load Balancer
Shared
Session
Web servers with WebFigure control
on page and WebFigure service
activated
Company
Boundary




## Manually Installing WebFigureService

`WebFigureService` is essentially a set of HTTP handlers that can service requests sent to an instance of Internet Information Service (IIS). There are occasions when you may want to manually install `WebFigureService`. For example:

- You want to implement the WebFigure controls programmatically and provide more detailed customization.
- Your Web environment was reconfigured from when you initially ran the "Quick Start: Implementing a WebFigure" on page 7-7.

- You want to implement WebFigures in a multiple server environment, as depicted in the previous graphic.
- You want to understand more about how WebFigures for .NET works.

When you dragged the GUI control for WebFigures onto the Web page in "Quick Start: Implementing a WebFigure" on page 7-7, you automatically installed `WebFigureService` in the Web application file `web.config`.

To install this manually:

1. Add a reference to `WebFiguresService.dll` from the folder *`InstallRoot`*`\toolbox\dotnetbuilder\bin\`*`arch`*`\v2.0` to the project, (where *`InstallRoot`* is the location of the installed MCR for machines with an installed MCR and *`matlabroot`* on a MATLAB Builder NE development machine without the MCR installed).

2. Add these lines to the `<httpHandlers>` section of `web.config`. This tells IIS to send any requests that come to the `__WebFigures.ashx` file to the `WebFigureHttpHandlerFactory` in the `WebFiguresService.dll`.

```
<httpHandlers>
        <add  path="__WebFigures.ashx"
                verb="GET"
                type="MathWorks.MATLAB.NET.WebFigures.
                        Service.Handlers.Factories.
                        Http.WebFigureHttpHandlerFactory"
                validate="false" />
</httpHandlers>
```

## Retrieving Multiple WebFigures From a Component

If your deployed component returns several WebFigures, then you have to make additional modifications to your code.

MATLAB sees a WebFigure the same way it see a `MWStructArray`. WebFigure constructors accept a WebFigure, an `MWArray`, or an `MWStructArray` as inputs.

Use the following examples as guides, depending on what type of functions you are working with.

### Working with Functions that Return a Single WebFigure as the Function's Only Output

#### C#

```
using MyComponent;
using MathWorks.MATLAB.NET.WebFigures;

public class
{
        protected void Page_Load(object sender, EventArgs e)
        {
                MyComponentclass myDeployedComponent =
                        new MyComponentclass();

                WebFigureControl1.WebFigure =
                        new WebFigure(myDeployedComponent.getKnot());
        }
}
```

#### Visual Basic

```
Imports MyComponent
Imports MathWorks.MATLAB.NET.WebFigures

Class
        Protected Sub Page_Load(ByVal sender As Object,
                                ByVal e As System.EventArgs)
                                Handles Me.Load
                Dim myDeployedComponent As _
                        New MyComponentclass()

                WebFigureControl1.WebFigure = _
                        New WebFigure(myDeployedComponent.getKnot())
        End Sub
End Class
```

### Working With Functions That Return Multiple WebFigures In an Array as the Output

#### C#

```
using MyComponent;
using MathWorks.MATLAB.NET.WebFigures;

public class
{
        protected void Page_Load(object sender, EventArgs e)
        {
                MyComponentclass myDeployedComponent =
                        new MyComponentclass();

   //If the function returns an array with 4 WebFigures
   //  in it and takes in no inputs.
   MWArray[] outputs = myDeployedComponent.getKnot(4);

                WebFigureControl1.WebFigure =
                        new WebFigure(outputs[0]);

   WebFigureControl2.WebFigure =
                        new WebFigure(outputs[1]);

   WebFigureControl3.WebFigure =
                        new WebFigure(outputs[2]);

     WebFigureControl4.WebFigure =
                        new WebFigure(outputs[3]);
        }
}
```

#### Visual Basic

```
Imports MyComponent
Imports MathWorks.MATLAB.NET.WebFigures

Class
        Protected Sub Page_Load(ByVal sender As Object,
```

```
                                ByVal e As System.EventArgs)
                                Handles Me.Load
                    Dim myDeployedComponent As _
                            New MyComponentclass()

    Dim outputs as MWArray() = _
    myDeployedComponent.getKnot(4)

                    WebFigureControl1.WebFigure = _
                            New WebFigure(outputs(0))

                    WebFigureControl2.WebFigure = _
                            New WebFigure(outputs(1))

                    WebFigureControl3.WebFigure = _
                            New WebFigure(outputs(2))

                    WebFigureControl4.WebFigure = _
                            New WebFigure(outputs(3))
            End Sub
    End Class
```

## Attaching a WebFigure

After you have manually installed `WebFigureService`, the server where it is installed is ready to receive requests for any `WebFigure` information. In the Quick Start, `WebFigureService` uses the session cache built into IIS to retrieve a `WebFigure`, per user, and display it. Since a `WebFigureControl` isn't being used in this case, you need to manually set up the `WebFigureService` and attach the `WebFigure`. Add the code supplied in this section to attach a `WebFigure` of your choosing.

This method of setting up `WebFigureService` and attaching the figure manually is very useful in the following situations:

- You do not want front-end servers to have `WebFigureService` running on them for performance reasons.
- You are displaying a `WebFigure` that does not change based on the current user or session. When multiple users are sharing the same `WebFigure`, which is very common, it is much more efficient to store a single `WebFigure`

in the `Application` or `Cache` state, rather than issuing all users their own figure.

There are a number of ways to attach a `WebFigure` to a scope, depending on state (note that these terms follow standard industry definitions and usage):

| State | Definition |
|---|---|
| `Session` | The method used by `WebFigureControl` by default, which is tied to a specific user session and cannot be shared across sessions. If you use IIS session sharing capabilities, you can use this across servers in a cluster. |
| `Application` | Available for any user of your application, per application lifetime. IIS will not propagate this across servers in a cluster, but if each server attaches the data to this cache once, all users can access it very efficiently. |
| `Cache` | Similar to `Application`, but with more potential settings. You can assign “time to live” and other settings found in Microsoft documentation. |

**Note** In this type of configuration, it is typical to have the following code executed once in the `Global.asax` server startup block. For more information on `Global.asax`, see “Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up” on page 7-28.

Add the following code to manually attach the `WebFigure`, based on whether you are using C# or Visual Basic:

- **C#:**

```
MyComponentclass myDeployedComponent =
        new MyComponentclass();

Session["SessionStateWebFigure"] =
        new WebFigure(myDeployedComponent.getKnot());
```

**Or**

```
Application["ApplicationStateWebFigure"] =
        new WebFigure(myDeployedComponent.getKnot());
```

**Or**

```
Cache["CacheStateWebFigure"] =
        new WebFigure(myDeployedComponent.getKnot());
```

- **Visual Basic:**

```
Dim myDeployedComponent As _
        New MyComponentclass()

Session("SessionStateWebFigure") = _
        New WebFigure(myDeployedComponent.getKnot())
```

**Or**

```
Application("ApplicationStateWebFigure") = _
        New WebFigure(myDeployedComponent.getKnot())
```

**Or**

```
Cache("CacheStateWebFigure") = _
        New WebFigure(myDeployedComponent.getKnot())
```

## Setting Up WebFigureControl for Remote Invocation

After you drag a `WebFigureControl` onto a page, as in "Quick Start: Implementing a WebFigure" on page 7-7, you either assign the `WebFigure` property or set the Remote Invocation properties, depending on how the figure will be used.

The procedure in this section allows you to tell `WebFigureControl` to reference a `WebFigure` that has been manually attached to a `WebFigureService` on a remote server or cluster of remote servers. This allows you to use the custom control, yet the resources of `WebFigureService` are running on a remote server to maximize performance.

1. Drag a `WebFigureControl` from the toolbox onto the page, if you haven't done so already in "Quick Start: Implementing a WebFigure" on page 7-7.

---

**Note** If you are running on a system with 64-bit architecture, use the information in "Advanced Configuration of a WebFigure" on page 7-15 to work with WebFigures unless you are deploying a Web site which is 32-bit only and you have a 32-bit MCR installed.

---

2. In the Properties pane for this control, set the **Name** and **Scope** attributes as follows:

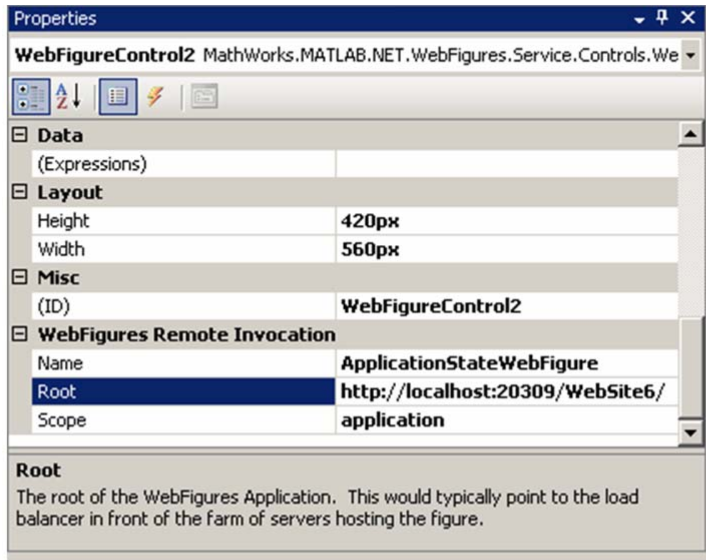
   - **Name** `ApplicationStateWebFigure`
   - **Scope** `application`

   ---

   **Caution** Always attempt to define the scope. If you leave **Scope** blank, the `Session` state, the `Application` state, and then `the Cache` state (in this order) will be checked. If there are `WebFigures` in any of these states with the same name, there can be potential for conflict and confusion. The first figure with the same name will be used by default.

   ---

   The pane should now look like this:




---

**Note** If you don't provide a **root** (usually the location of the load balancer), it is assumed to be the server where the page is executing.

---

## Getting an Embeddable String That References a WebFigure Attached to a WebFigureService

From any server, you can use the `GetHTMLEmbedString` API to get a string that can be embedded onto a page, if you followed the procedures "Manually Installing WebFigureService" on page 7-17 in "Attaching a WebFigure" on page 7-21.

To do so, use the following optional parameters and code snippets (or something similar, depending on your implementation). For information on the differences between session, application, and cache scopes, see "Attaching a WebFigure" on page 7-21.

### GetHTMLEmbedString API Parameters

| Parameter | If not specified... |
|---|---|
| `ID` | Default MATLAB WebFigure (the MATLAB membrane logo). |
| `Root` | The relative path to the current Web page will be used. |
| `WebFigureAttachType` | Will search through `Session` state, then `Application` state, then `Cache` state. |
| `Height` | Default height will be 420. |
| `Width` | Default width will be 560. |

## Referencing a WebFigure Attached to the Local Server

- **C#:**

```
using MathWorks.MATLAB.NET.WebFigures.Service;

String localEmbedString =
        WebFigureServiceUtility.GetHTMLEmbedString(
                "SessionStateWebFigure",
                WebFigureAttachType.session,
                300,
                300);

Response.Write(localEmbedString);
```

- **Visual Basic:**

```
Imports MathWorks.MATLAB.NET.WebFigures.Service

Dim localEmbedString As String = _
        WebFigureServiceUtility.GetHTMLEmbedString( _
                "SessionStateWebFigure", _
                WebFigureAttachType.session, _
                300, _
                300)
```

```
Response.Write(localEmbedString)
```

### Referencing a WebFigure Attached to a Remote Server

- **C#:**

```
using MathWorks.MATLAB.NET.WebFigures.Service;

String remoteEmbedString =
        WebFigureServiceUtility.GetHTMLEmbedString(
                "SessionStateWebFigure",
                "http://localhost:20309/WebSite7/",
                WebFigureAttachType.session,
                300,
                300);

Response.Write(remoteEmbedString);
```

- **Visual Basic:**

```
Imports MathWorks.MATLAB.NET.WebFigures.Service

Dim localEmbedString As String = _
        WebFigureServiceUtility.GetHTMLEmbedString( _
                "SessionStateWebFigure", _
                "http://localhost:20309/WebSite7/", _
                WebFigureAttachType.session, _
                300, _
                300)

Response.Write(localEmbedString)
```

## Improving Processing Times for JavaScript Using Minification

This application uses JavaScript to perform most of its AJAX functionality. Because JavaScript runs in the client browser, it must all be streamed to the client computer before it can execute. To improve this process, you use a standard JavaScript minification algorithm to remove comments and white

space in the code. This feature is enabled by default. To disable it, create an environment variable called mathworks.webfigures.disableJSMin and set its value to true.

## Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up

In ASP.NET there is a special type of object you can add called a *Global Application Class*, Also known by the name *Global.asax*.

Global.asax classes have methods that are called at various times in the IIS life cycle, such as Application_Start and Application_End. These methods get called respectively when the server is first started and when the server is being shut down.

As seen in "Quick Start: Implementing a WebFigure" on page 7-7, the default behavior for a WebFigureControl is to store data in the Session cache on the server. In other words, each user that accesses a page using a WebFigureControl has an individual instance of that WebFigure in the cache. This is useful if each user gets specific data, but resources can be wasted in situations where all users are accessing the same WebFigures.

Therefore, in order to maximize available resources, it makes sense to move WebFigure code for commonly used figures into the Application_Start method of the Global.asax. In the following example, code written in the Web page initialization section of "Attaching a WebFigure" on page 7-21 is moved into a Global.asax method as follows:

#### C#

```
void Application_Start(object sender, EventArgs e)
{
    // Code that runs on application startup
    MyComponentclass myDeployedComponent =
        new MyComponentclass();

    Application["ApplicationStateWebFigure"] =
        new WebFigure(myDeployedComponent.getKnot());

    //Or
```

```
        Cache["CacheStateWebFigure"] =
            new WebFigure(myDeployedComponent.getKnot());
    }
```

**Visual Basic**

```
    Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
            ' Code that runs on application startup
            Dim myDeployedComponent As _
                New MyComponentclass()


            Application("ApplicationStateWebFigure") = _
                New WebFigure(myDeployedComponent.getKnot())

            'Or

            Cache("CacheStateWebFigure") = _
                New WebFigure(myDeployedComponent.getKnot())
        End Sub
```

---

**Note** In this scenario, notice a WebFigure is not bound to the `Session`, since you usually need to share the WebFigures across different sessions. However, it may be useful to use the `Cache` option, since it provides a way to specify `Time To Live` so the WebFigure can be regenerated and reattached at a specific time interval.

Once the figure is attached to a cache, reference it either from the `WebFigureControl` as seen in "Setting Up WebFigureControl for Remote Invocation" on page 7-23 or directly from the Web page as in "Getting an Embeddable String That References a WebFigure Attached to a WebFigureService" on page 7-25.

---

# Upgrading Your WebFigures

If you want to upgrade your version of MATLAB Builder NE and retain WebFigures created with a prior product release, do the following:

1. Delete the `WebFigureControl` icon from the toolbox.
2. Delete any WebFigures from your page.
3. Upgrade your version of MATLAB Builder NE .
4. Add the new `WebFigureControl` icon to the toolbox.
5. Drag new WebFigures on to your page.

# Troubleshooting

Use the following section to diagnose error conditions encountered when implementing WebFigures for the .NET feature.

In WebFigures, there are two ways to display errors: by turning `debug` on for the site, and by turning it off. When `debug` is turned on, some error messages contain links to HTML pages that describe how the problem might be solved. When it is turned off, only the error message is shown.

Common causes of errors include:

- MCR is not installed or is the wrong version (meaning `MWArray.dll` is the wrong version or `WebFigureService.dll` is the wrong version).
- Deployed component is a different version than that compatible with the MCR.
- Incorrect framework is being used (only .NET 2.0 Framework is supported as of R2008b for WebFigures).
- `WebFigureService` is not installed. See "Manually Installing WebFigureService" on page 7-17.
- `WebFigure` is not attached to `WebFigureService`. See "Attaching a WebFigure" on page 7-21.
- Remote root URL is pointing to an invalid server.

Common errors and their diagnosis follow.

| Error | Diagnosis |
|---|---|
| Issue Displaying Image. Please Refresh. | Most often, this message is generated when the session state has expired and the WebFigure has been deleted. Refreshing the session will reestablish the WebFigure in cache and the figure will reappear. |
| No `WebFigure` Can Be Found with the Name Specified | The `WebFigure` isn't attached correctly. See "Attaching a WebFigure" on page 7-21. |

| Error | Diagnosis |
|---|---|
| `WebFigureService` Has Encountered an Unrecoverable Error | A critical error has occurred but the exact cause is unknown. Typically this is due to some type of system configuration issue that could not be anticipated. |
| `WebFigureService` Not Functioning | The `WebFigureService` `httpHanderFactory` could not be found on the server specified. See "Manually Installing WebFigureService" on page 7-17. |
| Could not find a part of the path *`pathname`* | The logging environment variable is set to a folder that does not exist. |

# Logging Levels

There are several logging levels that can be used to diagnose problems with WebFigures.

| Logging Level | Uses |
|---|---|
| `Severe` | Unrecoverable errors and exceptions |
| `Warning` | Recoverable errors that might occur |
| `Information` | Informative messages |
| `Finer` | For monitoring application flow (when different parts of an application are executed) |

You can manually set the log level by setting an environment variable called `mathworks.webfigures.logLevel` to one of the above strings.

If you set this environment variable to something other than the above strings or it is not set, it defaults to a level of `Warning` or `Severe` only.

By default, all exceptions are shown within the `WebFigure` control on the Web page when `debug` mode is on for the site.

If you want more detailed logging information, or log information when `debug` is not `on`, set an environment variable called `mathworks.webfigures.logLocation` to the location where the log file is written. The log file is named *yourwebappname*`WFSLog.txt`.

# 8

# Working with MATLAB Figures and Images


- “Your Role in Working with Figures and Images” on page 8-2
- “Creating and Modifying a MATLAB Figure” on page 8-3
- “Working with MATLAB Figure and Image Data” on page 8-6

# Your Role in Working with Figures and Images

When you work with figures and images as a MATLAB programmer, you are responsible for:

- Preparing a MATLAB figure for export
- Making changes to the figure (optional)
- Exporting the figure
- Cleaning up the figure window

When you work with figures and images as a front-end Web developer, some of the tasks you are responsible for include:

- Getting a WebFigure from a deployed component
- Getting raw image data from a deployed component converted into a byte array
- Getting a buffered image from a component
- Getting a buffered image or a byte array from a WebFigure

# Creating and Modifying a MATLAB Figure

**In this section...**


"Preparing a MATLAB Figure for Export" on page 8-3

"Changing the Figure (Optional)" on page 8-3

"Exporting the Figure" on page 8-4

"Cleaning Up the Figure Window" on page 8-4

"Example: Modifying and Exporting Figure Data" on page 8-5


### MATLAB Programmer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| **MATLAB programmer** | • MATLAB expert<br>• No IT experience<br>• No access to IT systems | • Develops models; implements in MATLAB<br>• Uses tools to create a component that is used by the .NET developer |

## Preparing a MATLAB Figure for Export

**1** Create a figure window. For example:

```
h = figure;
```

**2** Add graphics to the figure. For example:

```
surf(peaks);
```

## Changing the Figure (Optional)

Optionally, you can change the figure numerous ways. For example:

### Alter Visibility

```
set(h, 'Visible', 'off');
```

### Change Background Color

```
set(h, 'Color', [.8,.9,1]);
```

### Alter Orientation and Size

```
width=500;
height=500;
rotation=30;
elevation=30;
set(h, 'Position', [0, 0, width, height]);
view([rotation, elevation]);
```

## Exporting the Figure

Export the contents of the figure in one of two ways:

### WebFigure

To export as a WebFigure:

```
returnFigure = webfigure(h);
```

### Image Data

To export image data, for example:

```
imgform = 'png';
returnByteArray = figToImStream(`figHandle', h, ...
                  `imageFormat', imgForm, ...
                  `outputType', `uint8');
```

## Cleaning Up the Figure Window

To close the figure window:

```
close(h);
```

## Example: Modifying and Exporting Figure Data

### WebFigure

```
function returnFigure = getWebFigure()
h = figure;
set(h, 'Visible', 'off');
surf(peaks);
set(h, 'Color', [.8,.9,1]);
returnFigure = webfigure(h);
close(h);
```

### Image Data

```
function returnByteArray = getImageDataOrientation(height, width,
                                  elevation, rotation, imageFormat )
h = figure;
set(h, 'Visible', 'off');
surf(peaks);
set(h, 'Color', [.8,.9,1]);
set(h, 'Position', [0, 0, width, height]);
view([rotation, elevation]);
returnByteArray = figToImStream(`figHandle', h, ...
                                `imageFormat', imageFormat, ...
                                `outputType', `uint8');
close(h);
```

# Working with MATLAB Figure and Image Data

| In this section… |
|---|
| "For More Comprehensive Examples" on page 8-6 |
| "Working with Figures" on page 8-6 |
| "Working with Images" on page 8-7 |

### Front-End Web Developer

| | | |
|---|---|---|
| **Front-end Web developer** | • No MATLAB experience<br>• Minimal IT experience<br>• Expert at usability and Web page design<br>• Minimal access to IT systems<br>• Expert at ASPX | • As service consumer, manages presentation and usability<br>• Creates front-end applications<br>• Integrates MATLAB code with language-specific frameworks and environments<br>• Integrates WebFigures with the rest of the Web page |

## For More Comprehensive Examples

This section contains code snippets intended to demonstrate specific functionality related to working with figure and image data.

To see these snippets in the context of more fully-realized multi-step examples, see the *MATLAB Application Deployment Web Example Guide*.

## Working with Figures

### Getting a Figure From a Deployed Component

For information about how to retrieve a figure from a deployed component, see "Working with Functions that Return a Single WebFigure as the Function's Only Output" on page 7-19.

# Working with Images

## Getting Encoded Image Bytes from an Image in a Component

### .NET

```
public byte[] getByteArrayFromDeployedComponent()
{
    MWArray width = 500;
    MWArray height = 500;
    MWArray rotation = 30;
    MWArray elevation = 30;
    MWArray imageFormat = "png";

    MWNumericArray result =
        (MWNumericArray)deployment.getImageDataOrientation(
            height,
            width,
            elevation,
            rotation,
            imageFormat);
    return (byte[])result.ToVector(MWArrayComponent.Real);
}
```

## Getting a Buffered Image in a Component

### .NET

```
public byte[] getByteArrayFromDeployedComponent()
{
    MWArray width = 500;
    MWArray height = 500;
    MWArray rotation = 30;
    MWArray elevation = 30;
    MWArray imageFormat = "png";

    MWNumericArray result =
        (MWNumericArray)deployment.getImageDataOrientation(
```

```
                height,
                width,
                elevation,
                rotation,
                imageFormat);
        return (byte[])result.ToVector(MWArrayComponent.Real);
    }

    public Image getImageFromDeployedComponent()
    {
        byte[] byteArray = getByteArrayFromDeployedComponent();
        MemoryStream ms = new MemoryStream(myByteArray, 0,
        myByteArray.Length);
        ms.Write(myByteArray, 0, myByteArray.Length);
        return Image.FromStream(ms, true);
    }
```

## Getting Image Data from a WebFigure

The following example shows how to get image data from a WebFigure object. It also shows how to specify the image type and the orientation of the image.

### .NET

```
WebFigure figure =
    new WebFigure(deployment.getWebFigure());
WebFigureRenderer renderer =
    new WebFigureRenderer();

//Creates a parameter object that can be changed
// to represent a specific WebFigure and its orientation.
//If you dont set any values it uses the defaults for that
// figure (what they were when the figure was created in M).
WebFigureRenderParameters param =
    new WebFigureRenderParameters(figure);

param.Rotation = 30;
param.Elevation = 30;
```

```
param.Width = 500;
param.Height = 500;

//If you need a byte array that can be streamed out
// of a web page you can use this:
byte[] outputImageAsBytes =
    renderer.RenderToEncodedBytes(param);

//If you need a .NET Image (can't be used on the web)
// you can use this code:
Image outputImageAsImage =
    renderer.RenderToImage(param);
```

9

# Sharing Components Across Distributed Applications Using .NET Remoting


- "Overview" on page 9-2
- "Your Role in Building Distributed Applications" on page 9-4
- "Selecting the Best Method of Accessing Your Component: MWArray API or Native .NET API" on page 9-5
- "Creating a Remotable .NET Component" on page 9-7
- "Enabling Access to a Remotable .NET Component" on page 9-12

# Overview

**In this section...**

"What Are Remotable Components?" on page 9-2

"Benefits of Using .NET Remoting" on page 9-2

## What Are Remotable Components?

*Remotable .NET components* allow you to access MATLAB functionality remotely, as part of a distributed system consisting of multiple applications, domains, browsers, or machines.

To create a remotable component, you must first create the component and then enable others to access it.

## Benefits of Using .NET Remoting

There are many reasons to create remotable components:

- **Cost savings** — Changes to business logic do not require you to roll out new software to every client. Instead, you can confine new updates to a small set of business servers.
- **Increased security for Web applications** — Implementing .NET Remoting allows your database, for example, to reside safely behind one or more firewalls.
- **Software Compatibility** — Using remotable components, which employ standard formatting protocols like SOAP (Simple Object Access Protocol), can significantly enhance the compatibility of the component with libraries and applications.
- **Ability to run applications as Windows services** — To run as a Windows service, you must have access to a remoteable component hosted by the service. Applications implemented as a Windows service provide many benefits to application developers who require an automated server running as a background process independent of a particular user account.

- **Flexibility to isolate native code binaries that were previously incompatible** — Mix native and managed code (such as the MATLAB Compiler Runtime) without restrictions.

# Your Role in Building Distributed Applications

Depending on your role in your organization, you may need assistance to completely implement .NET Remoting. The next table, .NET Remoting Deployment Roles, Responsibilities, and Tasks, describes some of the different roles, or jobs, that MATLAB Builder NE users typically perform when designing, building, running, and deploying a remotable .NET component.

**.NET Remoting Deployment Roles, Responsibilities, and Tasks**

| Role | Goal | Tasks |
| --- | --- | --- |
| MATLAB programmer | Creates distributed .NET applications run by remotable components, from MATLAB code. | • Writes and deploys MATLAB code.<br>• Creates a deployable, remotable .NET component as in "Creating a Remotable .NET Component" on page 9-7. |
| .NET programmer | Exposes .NET applications to end users. | • Writes client/server code to access the remotable component as in "Using the MWArray API" on page 9-12 or "Using the Native .NET API: The Magic Square Example" on page 9-19. |

# Selecting the Best Method of Accessing Your Component: MWArray API or Native .NET API

As of R2008b, there are two data conversion API's that are available to marshal and format data across the managed (.NET) / unmanaged (MATLAB) code boundary. In addition to the previously available `MWArray` API, the new Native API is available. Each API has advantages and limitations and each has particular applications for which it is best suited.

The `MWArray` API, which consists of the `MWArray` class and several derived types that map to MATLAB data types, is the standard API that has been used since the introduction of MATLAB Builder NE. It provides full marshalling and formatting services for all basic MATLAB data types including sparse arrays, structures, and cell arrays. This API requires the MATLAB MCR to be installed on the target machine as it makes use of several primitive MATLAB functions. For information about using this API, see "Using the MWArray API" on page 9-12.

The Native API was designed especially, though not exclusively, to support .NET remoting. It allows you to pass arguments and return values using standard .NET types. This feature is especially useful for clients that access a remoteable component using the native interface API, as it does not require the client machine to have the MATLAB MCR installed. In addition, as only native .NET types are used in this API, there is no need to learn semantics of a new set of data conversion classes. This API does not directly support .NET analogs for the MATLAB structure and cell array types. For information about using this API, see "Using the Native .NET API: The Magic Square Example" on page 9-19.

### Features of the MWArray API Compared With the Native .NET API

| | MWArray API | Native .NET API |
|---|---|---|
| Marshalling/formatting for all basic MATLAB types | X | |
| Pass arguments and return values using standard .NET types | | X |

**Features of the MWArray API Compared With the Native .NET API (Continued)**

| | MWArray API | Native .NET API |
|---|---|---|
| Access to remotable component from client without installed MATLAB | | X |
| Access to remotable component from client without installed MCR (see "Using the Native .NET API: The Cell and Struct Example" on page 9-27). | | X |

## Using Native .NET Structure and Cell Arrays

MATLAB Builder NE's native .NET API accepts standard .NET data types for inputs and outputs to MATLAB function calls.

These standard .NET data types are wrapped by the `Object` class—the base class for all .NET data types. This object representation is sufficient as long as the MATLAB functions have numeric, logical, or string inputs or outputs. It does not work well for MATLAB-specific data types like structure (struct) and cell arrays, since the native representation of these arrays types result in a multi-dimensional `Object` array that is difficult to comprehend or process.

Instead, MATLAB Builder NE' provides a special class hierarchy for struct and cell array representation designed to easily interface with the native .NET API.

See "Using the Native .NET API: The Cell and Struct Example" on page 9-27 for details.

# Creating a Remotable .NET Component

**In this section...**

"Building a Remotable Component Using the Deployment Tool" on page 9-7

"Building a Remotable Component Using the mcc Command" on page 9-10

"Files Generated by the Compilation Process" on page 9-11

### MATLAB Programmer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| MATLAB programmer | • MATLAB expert<br>• No IT experience<br>• No access to IT systems | • Develops models; implements in MATLAB<br>• Uses tools to create a component that is used by the .NET developer |

## Building a Remotable Component Using the Deployment Tool

### Preparing to Build Your Remote Component with deploytool

1. Copy the example files as follows depending on whether you plan to use the `MWArray` API or the native .NET API:

   - **If using the MWArray API**, copy the following folder that ships with the MATLAB product to your working folder:

     ```
     matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET
     \MagicRemoteExample\MWArrayAPI\MagicSquareRemoteComp
     ```

     After you copy the files, at the MATLAB command prompt, change the working directory (`cd`) to the new `MagicSquareRemoteComp` subfolder in your working folder.

   - **If using the native .NET API**, copy the following folder that ships with the MATLAB product to your working folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VS8\NET\MagicRemoteExample\
NativeAPI\MagicSquareRemoteComp
```

After you copy the file, at the MATLAB command prompt, change the working directory (`cd`) to the new `MagicSquareRemoteComp` subfolder in your working folder.

**2** Write the MATLAB function Your MATLAB code does not require any additions to support .NET Remoting. The following code for the `makesquare` function is in the file `makesquare.m` in the `MagicSquareRemoteComp` subfolder:

```
function y = makesquare(x)

%MAKESQUARE Magic square of size x.
%   Y = MAKESQUARE(X) returns a magic square of size x.
%   This file is used as an example for the MATLAB
%   Builder NE product.

%   Copyright 2001-2010 The MathWorks, Inc.
%   $Revision: 1.1.4.32 $  $Date: 2010/02/12 05:12:39 $

    y = magic(x);
```

**3** In MATLAB, open the Deployment Tool by issuing the `deploytool` command.

**4** Click the **Actions** icon ( ) in the upper-right corner. Then, select **Settings**.

**a** Select **.NET** under **Project Settings**.

**b** Select **Enable .NET Remoting**.

**c** Click **OK**.

## Build Your Remote Component with deploytool

**1** Start MATLAB if you have not done so already.

**2** Type `deploytool` at the command prompt, and press **Enter**. The `deploytool` GUI opens.

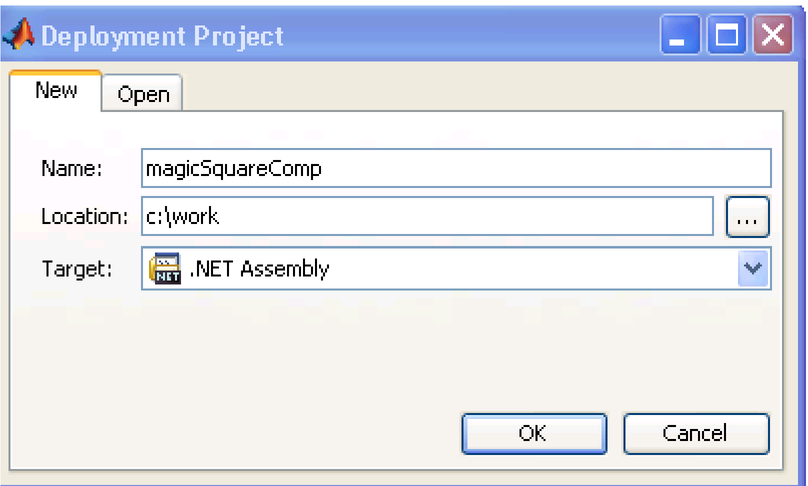
**3** Create a deployment project using the Deployment Project dialog box:

**a** Type the name of your project in the **Name** field.

**b** Enter the location of the project in the **Location** field. Alternately, navigate to the location.

**c** Select the target for the deployment project from the **Target** drop-down menu.

**d** Click **OK**.




**Creating a .NET Project**

**4** On the **Build** tab:

- If you are building a COM application, click **Add files**.
- If you are building a .NET application, click **Add class**. Type the name of the class in the Class Name field, designated by the letter c:

For this class, add methods you want to compile (your MATLAB by clicking **Add files**. To add another class, click **Add class**.

- You may optionally add supporting files. For examples of these files, see the `deploytool` Help. To add these files, in the Shared Resources and Helper Files area:

  e. Click **Add files/directories**

  f. Click **Open** to select the file or files.

5. When you complete your changes, click the Build button ( ).

## Building a Remotable Component Using the mcc Command

From the MATLAB prompt, issue the following command:

```
mcc -B
"dotnet:CompName,ClassName,FrameworkVersion,ShareFlag,RemoteFlag"
```

where:

- *`CompName`* is the name of the component you want to create.
- *`ClassName`* is the name of the C# class to which the component belongs.
- *`FrameworkVersion`* is the version of .NET Framework for the component you are building. For example, `2.0` would denote .NET Framework 2.0.
- *`ShareFlag`* designates access to the component. Values are either `private` or `shared`. Default is `private`.
- *`RemoteFlag`* designates either a remote or local component. Values are either `remote` or `local`. Default is `local`.

---

**Note** .NET Framework 1.1 is no longer supported as of R2008b.

---

For example, if you want to build a private remotable component using the Magic Square example in Chapter 1, "Getting Started", the `mcc` command to build the component for the .NET 2.0 Framework might look like this:

```
mcc -B "dotnet:MagicSquareComp,MagicSquareClass,2.0,private,remote"
```

## Files Generated by the Compilation Process

After compiling the components, ensure you have the following files in your `distrib` folder:

- `MagicSquareComp.dll` — The `MWArray` API component implementation assembly used by the server.
- `IMagicSquareComp.dll` — The `MWArray` API component interface assembly used by the client .
- `MagicSquareCompNative.dll` — The native .NET API component implementation assembly used by the server.
- `IMagicSquareCompNative.dll` — The native .NET API component interface assembly used by the client. You do not need to install an MCR on the client when using this interface.

# Enabling Access to a Remotable .NET Component

**In this section...**


"Using the MWArray API" on page 9-12

"Using the Native .NET API: The Magic Square Example" on page 9-19

"Using the Native .NET API: The Cell and Struct Example" on page 9-27


### .NET Developer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| **.NET Developer** | • Little to no MATLAB experience<br>• Moderate IT experience<br>• .NET expert<br>• Minimal access to IT systems | • Integrates deployed component with the rest of the .NET application |

## Using the MWArray API

### Why Use the MWArray API?

After you create the remotable component, you can set up a console server and client using the `MWArray` API. For more information on choosing the right API for your access needs, see "Selecting the Best Method of Accessing Your Component: MWArray API or Native .NET API" on page 9-5.

Some reasons you might use the `MWArray` API instead of the native .NET API are:

- You are working with data structure arrays, which the native .NET API does not support.
- You or your users work extensively with many MATLAB data types.
- You or your users are familiar and comfortable using the `MWArray` API.

For information on accessing your component using the native .NET API, see "Using the Native .NET API: The Magic Square Example" on page 9-19.

### Coding and Building the Hosting Server Application and Configuration File

The server application hosts the remote component built in "Creating a Remotable .NET Component" on page 9-7. You can also perform these steps using the MWArray API (see "Using the Native .NET API: The Magic Square Example" on page 9-19).

The client application, running in a separate process, accesses the remote component hosted by the server application. Build the server using the Microsoft Visual Studio project file `MagicSquareServer\MagicSquareMWServer.csproj`:

1. Change the references for the generated component assembly to `MagicSquareComp\distrib\MagicSquareComp.dll`.
2. Select the appropriate build platform.
3. Select **Debug** or **Release** mode.
4. Build the `MagicSquareMWServer` project.
5. Supply the configuration file for the `MagicSquareMWServer`.

**MagicSquareServer Code.** Use the C# code for the server located in the file `MagicSquareServer\MagicSquareServer.cs`:

```
using System;
using System.Runtime.Remoting;

namespace MagicSquareServer
  {
    class MagicSquareServer
      {
        static void Main(string[] args)
          {
            RemotingConfiguration.Configure
              (@"..\..\..\..\MagicSquareServer.exe.config");
```

```
            Console.WriteLine("Magic Square Server started...");

            Console.ReadLine();
          }
      }
  }
```

This code does the following processing:

- Reads the associated configuration file to determine
  - The name of the component that it will host
  - The remoting protocol and message formatting to use
  - The lease time for the remote component
- Signals that the server is active and waits for a carriage return to be entered before terminating.

**MagicSquareServer Configuration File.** The configuration file for the `MagicSquareServer` is in the file `MagicSquareServer\MagicSquareServer.exe.config`. The entire configuration file, written in XML, follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
                   type="MagicSquareComp.MagicSquareClass, MagicSquareComp"
                   objectUri="MagicSquareClass.remote" />
      </service>
      <lifetime leaseTime= "5M" renewOnCallTime="2M"
                leaseManagerPollTime="10S" />
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
```

```
          </channel>
        </channels>
      </application>
      <debug loadTypes="true"/>
    </system.runtime.remoting>
  </configuration>
```

This code specifies:

- The mode in which the remote component will be accessed—in this case, single call mode
- The name of the remote component, the component assembly, and the object URI (uniform resource identifier) used to access the remote component
- The lease time for the remote component
- The remoting protocol (`TCP/IP`) and port number
- The message formatter (`binary`) and the permissions for the communication channel (`full` trust)
- The server debugging option

### Coding and Building the Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application you built previously. (See "Coding and Building the Hosting Server Application and Configuration File" on page 9-13.

Next build the remote client using the Microsoft Visual Studio project file `MagicSquareClient\MagicSquareMWClient.csproj`. This file references both the shared data conversion assembly *`matlabroot`*`\toolbox\dotnetbuilder\bin\win32\v2.0\MWArray.dll` and the generated component interface assembly `MagicSquareComp\distrib\IMagicSquareComp`.

To create the remote client using Microsoft Visual Studio:

1. Select the appropriate build platform.

2 Select **Debug** or **Release** mode.

3 Build the `MagicSquareMWClient` project.

4 Supply the configuration file for the `MagicSquareMWServer`.

**MagicSquareClient Code.** Use the C# code for the client located in the file `MagicSquareClient\MagicSquareClient.cs`. The client code is shown here:

```
using System;
using System.Configuration;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

using System.Collections;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Remoting.Channels.Tcp;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using IMagicSquareComp;

namespace MagicSquareClient
{
  class MagicSquareClient
    {
      static void Main(string[] args)
        {
          try
            {
              RemotingConfiguration.Configure
                                        (@"MagicSquareClient.exe.config");

              String urlServer=
                   ConfigurationSettings.AppSettings["MagicSquareServer"];

              IMagicSquareClass magicSquareComp=
                (IMagicSquareClass)Activator.GetObject
```

```
                                    (typeof(IMagicSquareClass),
                                     urlServer);

                // Get user specified command line arguments or set default
                double arraySize= (0 != args.Length)
                                  ? Double.Parse(args[0]) : 4;

                // Compute the magic square and print the result
                MWNumericArray magicSquare=
                  (MWNumericArray)magicSquareComp.makesquare
                                    (arraySize);

                Console.WriteLine("Magic square of order {0}\n\n{1}",
                                  arraySize, magicSquare);
              }

            catch (Exception exception)
              {
                Console.WriteLine(exception.Message);
              }

            Console.ReadLine();
          }
      }
  }
```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remoteable component.
- The client instantiates the remoteable object using the static `Activator.GetObject` method
- From this point, the remoting client calls methods on the remoteable component exactly as it would call a local component method.

**MagicSquareClient Configuration File.** The configuration file for the magic square client is in the file `MagicSquareClient\MagicSquareClient.exe.config`. The configuration file, written in XML, is shown here:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MagicSquareServer"
     value="tcp://localhost:1234/MagicSquareClass.remote"/>
  </appSettings>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel name="MagicSquareChannel" ref="tcp" port="0">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

This code specifies:

- The name of the remote component server and the remote component URI (uniform resource identifier)
- The remoting protocol (`TCP/IP`) and port number
- The message formatter (`binary`) and the permissions for the communication channel (`full` trust)

### Starting the Server Application

Starting the server by doing the following:

1. Open a DOS or UNIX command window and `cd` to `MagicSquareServer\bin\x86\v2.0\Debug`.
2. Run `MagicSquareServer.exe`. You will see the message:

```
Magic Square Server started...
```

### Starting the Client Application

Start the client by doing the following:

1. Open a DOS or UNIX command window and `cd` to `MagicSquareClient\bin\x86\v2.0\Debug`.

2. Run `MagicSquareClient.exe`. After the MCR initializes, you should see the following output:

```
Magic square of order 4

16  2  3 13
 5 11 10  8
 9  7  6 12
 4 14 15  1
```

## Using the Native .NET API: The Magic Square Example

### .NET Developer

| Role | Knowledge Base | Responsibilities |
|---|---|---|
| .NET Developer | • Little to no MATLAB experience<br>• Moderate IT experience<br>• .NET expert<br>• Minimal access to IT systems | • Integrates deployed component with the rest of the .NET application |

### Why Use the Native .NET API?

After the remotable component has been created, you can set up a server application and client using the native .NET API. For more information on

choosing the right API for your access needs, see "Selecting the Best Method of Accessing Your Component: MWArray API or Native .NET API" on page 9-5.

Some reasons you might use the native .NET API instead of the `MWArray` API are:

- You want to pass arguments and return values using standard .NET types, and you or your users don't work extensively with data types specific to MATLAB.
- You want to access your component from a client machine without an installed version of MATLAB.

For information on accessing your component using the `MWArray` API, see "Using the MWArray API" on page 9-12.

### Coding and Building the Hosting Server Application and Configuration File

The server application will host the remote component you built in "Creating a Remotable .NET Component" on page 9-7.

The client application, running in a separate process, will access the remote component hosted by the server application. Build the server with the Microsoft Visual Studio project file `MagicSquareServer\MagicSquareMWServer.csproj`:

1. Change the references for the generated component assembly to `MagicSquareComp\distrib\MagicSquareCompNative.dll`.
2. Select the appropriate build platform (32-bit or 64-bit).
3. Select **Debug** or **Release** mode.
4. Build the `MagicSquareServer` project.
5. Supply the configuration file for the `MagicSquareServer`.

**MagicSquareServer Code.** TheC# code for the server isin the file `MagicSquareServer\MagicSquareServer.cs`. The `MagicSquareServer.cs` server code is shown here:

```
using System;
   using System.Runtime.Remoting;

   namespace MagicSquareServer
     {
       class MagicSquareServer
         {
           static void Main(string[] args)
             {
               RemotingConfiguration.Configure
                 (@"..\..\..\..\MagicSquareServer.exe.config");

               Console.WriteLine("Magic Square Server started...");

               Console.ReadLine();
             }
         }
     }
```

This code does the following:

- Reads the associated configuration file to determine the name of the component that it will host, the remoting protocol and message formatting to use, as well as the lease time for the remote component.
- Signals that the server is active and waits for a carriage return to be entered before terminating.

**MagicSquareServer Configuration File.** The configuration file for the `MagicSquareServer` is in the file `MagicSquareServer\MagicSquareServer.exe.config`. The entire configuration file, written in XML, is shown here:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
                   type="MagicSquareCompNative.MagicSquareClass,
                         MagicSquareCompNative"
```

```
                    objectUri="MagicSquareClass.remote" />
      </service>
      <lifetime leaseTime= "5M" renewOnCallTime="2M"
                leaseManagerPollTime="10S" />
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
    <debug loadTypes="true"/>
  </system.runtime.remoting>
   </configuration>
```

This code specifies:

- The mode in which the remote component will be accessed—in this case, single call mode
- The name of the remote component, the component assembly, and the object URI (uniform resource identifier) used to access the remote component
- The lease time for the remote component
- The remoting protocol (`TCP/IP`) and port number
- The message formatter (`binary`) and the permissions for the communication channel (`full` trust)
- The server debugging option

### Coding and Building the Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application built in "Coding and Building the Hosting Server Application and Configuration File" on page 9-20. Build the remote client using the Microsoft Visual Studio project file `MagicSquareClient\MagicSquareClient.csproj` which references both the shared data conversion assembly

*matlabroot*\toolbox\dotnetbuilder\bin\win32\v2.0\
MWArray.dll and the generated component interface assembly
MagicSquareComp\distrib\IMagicSquareCompNative. To create the remote client using Microsoft Visual Studio:

1. Select the appropriate build platform.
2. Select **Debug** or **Release** mode.
3. Build the MagicSquareClient project.
4. Supply the configuration file for the MagicSquareServer.

**MagicSquareClient Code.** TheC# code for the client isin the file MagicSquareClient\MagicSquareClient.cs. The client code is shown here:

```
using System;
using System.Configuration;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

using System.Collections;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Remoting.Channels.Tcp;

using IMagicSquareCompNative;

namespace MagicSquareClient
{
  class MagicSquareClient
    {
      static void Main(string[] args)
        {
          try
            {
              RemotingConfiguration.Configure
                (@"MagicSquareClient.exe.config");

              String urlServer=
                ConfigurationSettings.AppSettings["MagicSquareServer"];
```

```
                IMagicSquareClassNative magicSquareComp=
                  (IMagicSquareClassNative)Activator.GetObject
                    (typeof(IMagicSquareClassNative), urlServer);

                // Get user specified command line arguments or set default
                double arraySize= (0 != args.Length)
                                  ? Double.Parse(args[0]) : 4;

                // Compute the magic square and print the result
                double[,] magicSquare=
                            (double[,])magicSquareComp.makesquare(arraySize);

                Console.WriteLine("Magic square of order {0}\n", arraySize);

                // Display the array elements:
                for (int i = 0; i < (int)arraySize; i++)
                  for (int j = 0; j < (int)arraySize; j++)
                    Console.WriteLine
                              ("Element({0},{1})= {2}", i, j, magicSquare[i, j]);
              }

            catch (Exception exception)
              {
                Console.WriteLine(exception.Message);
              }

            Console.ReadLine();
          }
      }
  }
```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remoteable component.
- The client instantiates the remoteable object using the static `Activator.GetObject` method

- From this point, the remoting client calls methods on the remoteable component exactly as it would call a local component method.

**MagicSquareClient Configuration File.** The configuration file for the magic square client is in the file `MagicSquareClient\MagicSquareClient.exe.config`. The configuration file, written in XML, is shown here:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MagicSquareServer"
            value="tcp://localhost:1234/MagicSquareClass.remote"/>
  </appSettings>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel name="MagicSquareChannel" ref="tcp" port="0">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

This code specifies:

- The name of the remote component server and the remote component URI (uniform resource identifier)
- The remoting protocol (`TCP/IP`) and port number
- The message formatter (`binary`) and the permissions for the communication channel (`full` trust)

### Starting the Server Application

Start the server by doing the following:

1 Open a DOS or UNIX command and `cd` to `MagicSquareServer\bin\x86\v2.0\Debug`.

2 Run `MagicSquareServer.exe`. You will see the message:

```
Magic Square Server started...
```

### Starting the Client Application

Start the client by doing the following:

1 Open a DOS or UNIX command window and `cd`to `MagicSquareClient\bin\x86\v2.0\Debug`.

2 Run `MagicSquareClient.exe`. After the MCR initializes you should see the following output:

```
Magic square of order 4

Element(0,0)= 16
Element(0,1)= 2
Element(0,2)= 3
Element(0,3)= 13
Element(1,0)= 5
Element(1,1)= 11
Element(1,2)= 10
Element(1,3)= 8
Element(2,0)= 9
Element(2,1)= 7
Element(2,2)= 6
Element(2,3)= 12
Element(3,0)= 4
Element(3,1)= 14
Element(3,2)= 15
Element(3,3)= 1
```

# Using the Native .NET API: The Cell and Struct Example

## Why Use the .NET API With Cell Arrays and Structs?

Using .NET representations of MATLAB struct and cell arrays is recommended if both of these are true:

- You have MATLAB functions on a server with MATLAB struct or cell data types as inputs or outputs
- You do not want or need to install an MCR on your client machines

The native `MWArray`, `MWStructArray`, and `MWCellArray` classes are members of the `MathWorks.MATLAB.NET.Arrays.native` namespace.

The class names in this namespace are identical to the class names in the `MathWorks.MATLAB.NET.Arrays`. The difference is that the native representation of struct and cell arrays have no methods or properties that require an MCR.

The *`matlabroot`*`\toolbox\dotnetbuilder\Examples\VS8\NET` folder has example solutions you can practice building. The `NativeStructCellExample` folder contains native struct and cell examples.

## Building Your Component

This example demonstrates how to deploy a remotable component using native struct and cell arrays. Before you set up the remotable client and server code, build a remotable component.

If you have not yet built the component you want to deploy, see the instructions in “Building a Remotable Component Using the Deployment Tool” on page 9-7 or “Building a Remotable Component Using the mcc Command” on page 9-10.

## The Native .NET Cell and Struct Example

The server application hosts the remote component.

The client application, running in a separate process, accesses the remote component hosted by the server application. Build the server with the Microsoft Visual Studio project file `NativeStructCellServer.csproj`:

1 Change the references for the generated component assembly to *component_name*`\distrib\`*component_name*`Native.dll`.

2 Select the appropriate build platform.

3 Select **Debug** or **Release** mode.

4 Build the `NativeStructCellServer` project.

5 Supply the configuration file for the `NativeStructCellServer`. The C# code for the server is in the file `NativeStructCellServer.cs`:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;

namespace NativeStructCellServer
{
    class NativeStructCellServer
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure(@"NativeStructCellServer.exe.config");

            Console.WriteLine("NativeStructCell Server started...");

            Console.ReadLine();
        }
    }
}
```

This code reads the associated configuration file to determine:

- Name of the component to host
- Remoting protocol and message formatting to use

- Lease time for the remote component

In addition, the code also signals that the server is active and waits for a carriage return before terminating.

### Coding and Building the Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application built in "The Native .NET Cell and Struct Example" on page 9-27. Build the remote client using the Microsoft Visual Studio project file `NativeStructCellClient\NativeStructCellClient.csproj` which references both the shared data conversion assembly `*matlabroot*\toolbox\dotnetbuilder\bin\win32\v2.0\MWArray.dll` and the generated component interface assembly `*component_name*\distrib\I*component_name*Native`. To create the remote client using Microsoft Visual Studio:

1. Select the appropriate build platform.
2. Select **Debug** or **Release** mode.
3. Build the `NativeStructCellClient` project.
4. Supply the configuration file for the `NativeStructCellClient`.

**NativeStructCellClient Code.** TheC# code for the client is in the file `NativeStructCellClient\NativeStructCellClient.cs`:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Configuration;
using MathWorks.MATLAB.NET.Arrays.native;

using INativeStructCellCompNative;

// This is a simple example that demonstrates the use
// of MathWorks.MATLAB.NET.Arrays.native package.
namespace NativeStructCellClient
```

```
{
    class NativeStructCellClient
    {
        static void Main(string[] args)
        {
            try
            {
                RemotingConfiguration.Configure(@"NativeStructCellClient.exe.config");
                String urlServer =
                         ConfigurationSettings.AppSettings["NativeStructCellServer"];
                INativeStructCellClassNative nativeStructCell =
                         (INativeStructCellClassNative)Activator.GetObject(typeof
                                           (INativeStructCellClassNative), urlServer);

                MWCellArray field_names = new MWCellArray(1, 2);
                field_names[1, 1] = "Name";
                field_names[1, 2] = "Address";

                Object[] o = nativeStructCell.createEmptyStruct(1,field_names);
                MWStructArray S1 = (MWStructArray)o[0];
                Console.WriteLine("\nEVENT 2: Initialized structure as
                             received in client applications:\n\n{0}" , S1);

                //Convert "Name" value from char[,] to a string since there's no
                                MWCharArray constructor on server that accepts
                //char[,] as input.
                char c = ((char[,])S1["Name"])[0, 0];
                S1["Name"] = c.ToString();

                MWStructArray address = new MWStructArray(new int[] { 1, 1 },
                                    new String[] { "Street", "City", "State", "Zip" });
                address["Street", 1] = "3, Apple Hill Drive";
                address["City",   1] = "Natick";
                address["State",  1] = "MA";
                address["Zip",    1] = "01760";

                Console.WriteLine("\nUpdating the 'Address' field to :\n\n{0}", address);
                Console.WriteLine("\n#################################\n");
                S1["Address",1] = address;
```

```
                Object[] o1 = nativeStructCell.updateField(1, S1, "Name");
                MWStructArray S2 = (MWStructArray)o1[0];

                Console.WriteLine("\nEVENT 5: Final structure as
                                              received by client:\n\n{0}" , S2);
                Console.WriteLine("\nAddress field: \n\n{0}" , S2["Address",1]);
                Console.WriteLine("\n##################################\n");
            }
            catch (Exception exception)
            {
                Console.WriteLine(exception.Message);
            }
            Console.ReadLine();
        }
    }
}
```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remoteable component.
- The client instantiates the remoteable object using the static `Activator.GetObject` method
- From this point, the remoting client calls methods on the remoteable component exactly as it would call a local component method.

**NativeStructCellClient Configuration File.** The configuration file for the `NativeStructCellClient` is in the file `NativeStructCellClient\NativeStructCellClient.exe.config`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="NativeStructCellServer" value=
              "tcp://localhost:1236/NativeStructCellClass.remote"/>
  </appSettings>
  <system.runtime.remoting>
    <application>
      <channels>
```

```
            <channel name="NativeStructCellChannel" ref="tcp" port="0">
              <clientProviders>
                <formatter ref="binary" />
              </clientProviders>
              <serverProviders>
                <formatter ref="binary" typeFilterLevel="Full" />
              </serverProviders>
            </channel>
          </channels>
        </application>
      </system.runtime.remoting>
    </configuration>
```

This code specifies:

- Name of the remote component server and the remote component URI (uniform resource identifier)
- Remoting protocol (`TCP/IP`) and port number
- Message formatter (`binary`) and the permissions for the communication channel (`full` trust)

### Starting the Server Application

Start the server by doing the following:

1. Open a DOS or UNIX command window and `cd` to `NativeStructCellServer\bin\x86\v2.0\Debug`.
2. Run `NativeStructCellServer.exe`. The following output appears:

```
EVENT 1: Initializing the structure on server and sending
          it to client:
         Initialized empty structure:

       Name: ' '
    Address: []


##################################
```

```
EVENT 3: Partially initialized structure as
         received by server:

      Name: ' '
   Address: [1x1 struct]

Address field as initialized from the client:

   Street: '3, Apple Hill Drive'
     City: 'Natick'
    State: 'MA'
      Zip: '01760'

##################################

EVENT 4: Updating 'Name' field before sending the
         structure back to the client:

      Name: 'The MathWorks'
   Address: [1x1 struct]


##################################
```

### Starting the Client Application

Start the client by doing the following:

1 Open a DOS or UNIX command windowand `cd` to `NativeStructCellClient\bin\x86\v2.0\Debug`.

2 Run `NativeStructCellClient.exe`. After the MCR initializes, the following output appears:

```
EVENT 2: Initialized structure as received in client applications:

1x1 struct array with fields:
       Name
       Address
```

```
Updating the 'Address' field to :

1x1 struct array with fields:
       Street
       City
       State
       Zip

#################################


EVENT 5: Final structure as received by client:

1x1 struct array with fields:
       Name
       Address

Address field:

1x1 struct array with fields:
       Street
       City
       State
       Zip

#################################
```

### Coding and Building the Client Application and Configuration File with the Native MWArray, MWStructArray, and MWCellArray Classes

**createEmptyStruct.m.** Initialize the structure on the server and send it to the client with the following MATLAB code:

```
function PartialStruct = createEmptyStruct(field_names)

fprintf('EVENT 1: Initializing the structure on server
```

```
                              and sending it to client:\n');

PartialStruct = struct(field_names{1},' ',field_names{2},[]);

fprintf('         Initialized empty structure:\n\n');
disp(PartialStruct);
fprintf('\n###################################\n');
```

**updateField.m.** Receive the partially updated structure from the clientand addmore data to it, before passing it back to the client, with the following MATLAB code:

```
function FinalStruct = updateField(st,field_name)

fprintf('\nEVENT 3: Partially initialized structure as
                              received by server:\n\n');
disp(st);
fprintf('Address field as initialized from the client:\n\n');
disp(st.Address);
fprintf('##################################\n');

fprintf(['\nEVENT 4: Updating ''', field_name, '''
   field before sending the structure back to the client:\n\n']);
st.(field_name) = 'The MathWorks';
FinalStruct = st;
disp(FinalStruct);
fprintf('\n###################################\n');
```

**NativeStructCellClient.cs.** Create the client C# code:

---

**Note** In this case, you do not need the MCR on the system path.

---

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Configuration;
using MathWorks.MATLAB.NET.Arrays.native;
```

```
using INativeStructCellCompNative;

// This is a simple example that demonstrates the use of
//     MathWorks.MATLAB.NET.Arrays.native package.
namespace NativeStructCellClient
{
    class NativeStructCellClient
    {
        static void Main(string[] args)
        {
            try
            {
                RemotingConfiguration.Configure
                      (@"NativeStructCellClient.exe.config");
                String urlServer =
                      ConfigurationSettings.AppSettings["NativeStructCellServer"];
                INativeStructCellClassNative nativeStructCell =
                      (INativeStructCellClassNative)Activator.GetObject(typeof
                           (INativeStructCellClassNative),
                           urlServer);

                MWCellArray field_names = new MWCellArray(1, 2);
                field_names[1, 1] = "Name";
                field_names[1, 2] = "Address";

                Object[] o = nativeStructCell.createEmptyStruct(1,field_names);
                MWStructArray S1 = (MWStructArray)o[0];
                Console.WriteLine("\nEVENT 2: Initialized structure as received
                                     in client applications:\n\n{0}" , S1);

                //Convert "Name" value from char[,] to a string since
                // there's no MWCharArray constructor
                // on server that accepts char[,] as input.
                char c = ((char[,])S1["Name"])[0, 0];
                S1["Name"] = c.ToString();

                MWStructArray address =
                      want new MWStructArray(new int[] { 1, 1 },
                             new String[] { "Street", "City", "State", "Zip" });
```

```
                address["Street", 1] = "3, Apple Hill Drive";
                address["City",   1] = "Natick";
                address["State",  1] = "MA";
                address["Zip",    1] = "01760";

                Console.WriteLine("\nUpdating the
                                   'Address' field to :\n\n{0}", address);
                Console.WriteLine("\n#################################\n");
                S1["Address",1] = address;

                Object[] o1 = nativeStructCell.updateField(1, S1, "Name");
                MWStructArray S2 = (MWStructArray)o1[0];

                Console.WriteLine("\nEVENT 5: Final structure as received by
                                     client:\n\n{0}" , S2);
                Console.WriteLine("\nAddress field: \n\n{0}" , S2["Address",1]);
                Console.WriteLine("\n#################################\n");
            }
            catch (Exception exception)
            {
                Console.WriteLine(exception.Message);
            }
            Console.ReadLine();
        }
    }
}
```

**NativeStructCellServer.cs.** Create the server C# code:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;

namespace NativeStructCellServer
{
    class NativeStructCellServer
    {
        static void Main(string[] args)
        {
```

```
                RemotingConfiguration.Configure(@"NativeStructCellServer.exe.config");

                Console.WriteLine("NativeStructCell Server started...");

                Console.ReadLine();
            }
        }
    }
```

# 10

# Troubleshooting

This chapter provides some solutions to common problems encountered using the MATLAB Builder NE product.


- "Troubleshooting the Build Process " on page 10-2
- "Failure to Find a Required File" on page 10-3
- "Diagnostic Messages" on page 10-4

# Troubleshooting the Build Process

| In this section... |
|---|
| "Viewing the Latest Build Log" on page 10-2 |
| "Generating Verbose Output" on page 10-2 |

## Viewing the Latest Build Log

To view the log of your most recent build process, open the build log, which is generated in the intermediate folder for your project. By default, the intermediate folder for a project is *`project_folder/projectname_without_ext/src`*.

## Generating Verbose Output

Telling the Deployment Tool to generate verbose output provides a more detailed log of each build. These details can assist you in determining the cause of problems you encounter.

To enable verbose output during builds, select **Generate Verbose Output** in the Deployment Tool window.

# Failure to Find a Required File

If your application generates a diagnostic message indicating that a module cannot be found, it could be that the MCR is not located properly on your path. How to fix this problem depends on whether it occurs on a development machine (where you are using the builder to create a component) or target machine (where you are trying to use the component in your application). The required locations are as follows for the MCR according to development versus target machines.

- Make sure that *`matlabroot`*`\runtime\`*`architecture`* appears on your system path ahead of any other MATLAB installations. (*`matlabroot`* is your root MATLAB folder.)
- Verify that *`mcr_root`*`\`*`ver`*`\runtime`
  `\`*`architecture`* appears on your system path.
  (*`mcr_root`* is your root MCR folder) and *`ver`* represents the MCR version number.

# Diagnostic Messages

The following table shows diagnostic messages you might encounter, probable causes for the message, and suggested solutions.

---

**Note** The MATLAB Builder NE product uses the MATLAB Compiler product to generate components. This means that you might see diagnostic messages from MATLAB Compiler. See "Compile-Time Errors" in the MATLAB Compiler documentation for more information about those messages.

---

See the following table for information about some diagnostic messages.

### Diagnostic Messages and Suggested Solutions

| Message | Probable Cause | Suggested Solution |
|---|---|---|
| `LoadLibrary ("`*`component_name`*`_1_0.dll") failed - The specified module could not be found.` | You may get this error message while registering the project DLL from the DOS prompt. This usually occurs if the MATLAB product is not on the system path. | See "Failure to Find a Required File" on page 10-3. |
| | You might also get this error if you try to deploy your component without adding the path for the DLL to the system path on the target machine. | On the target machine where the COM component is to be used:<br>**1** Use the `extractCTF.exe` utility to decompress the `.ctf` file generated by the builder when you built the COM component.<br>**2** Look at the files in the CTF, and note the path for the DLL.<br>**3** Add this path to the system path. |

**Diagnostic Messages and Suggested Solutions (Continued)**

| Message | Probable Cause | Suggested Solution |
|---|---|---|
| | | See the MATLAB Compiler documentation for more information about `extractctf.exe`. |
| `MBUILD.BAT: Error: The chosen compiler does not support building COM objects.` | The chosen compiler does not support building COM objects. | Rerun `mbuild -setup` and choose a supported compiler. |
| `Error in` *`component_name`*`.`*`class_name`*`.x: Error getting data conversion flags.` | This is often caused by `mwcomutil.dll` not being registered. | **1** Open a DOS window.<br>**2** Change folders to *`matlabroot`*`\runtime\`*`architecture`*.<br>**3** Run the following command: `mwregsvr mwcomutil.dll`<br>(*`matlabroot`* is your root MATLAB folder.) |
| `Error in VBAProject: ActiveX component can't create object.` | • Project DLL is not registered.<br>• An incompatible MATLAB DLL exists somewhere on the system path. | If the DLL is not registered,<br>**1** Open a DOS window.<br>**2** Change folders to *`projectdir`*`\distrib`.<br>**3** Run the following command: `mwregsvr` *`projectdll`*`.dll`<br>(*`projectdir`* represents the location of your project files). |

**Diagnostic Messages and Suggested Solutions (Continued)**

| Message | Probable Cause | Suggested Solution |
|---|---|---|
| `object ref not set to instance of an object` | This occurs when an object that has not been instantiated is called | Instantiate the object (declare it as `new`). See "Classes and Methods" on page 4-7in this User's Guide for more information. |
| `Error in VBAProject: Automation error The specified module could not be found.` | This usually occurs if MATLAB is not on the system path. | See "Failure to Find a Required File" on page 10-3. |
| `QueryInterface for interface <`*`COM OBJECT NAME`*`> failed.` | You might be using the incorrect number and/or type of function parameters to call into your COM object. | Function calls to COM objects that encapsulate MATLAB functions must have the same number and data type of arguments as the COM object. In general:<br>• Use a Variant data type for the return type of the COM object.<br>• Use doubles as default numeric input parameters (rather than integers).<br>You might also use development tools such as OLEVIEW and Object Browser, which ship with Microsoft Visual Studio and Microsoft Visual Basic, respectively, to verify the expected function signature of TypeLib for the COM object. |
| `Showing a modal dialog box or form when the application is not running in UserInteractive` | This warning occurs when ASP.NET code tries to bring up a dialog box. | Work around this problem by doing the following:<br>1 Open the Windows Control Panel. |

**Diagnostic Messages and Suggested Solutions (Continued)**

| Message | Probable Cause | Suggested Solution |
|---|---|---|
| `mode is not a valid operation.  Specify the ServiceNotification or DefaultDesktopOnly style to display a notification from a service application.` | If occurs because `getframe()` makes the figure window visible before performing the capture and thus fails when running in IIS. `msgbox()` calls in MATLAB code cause the warning to appear also.. | **2** Open Services.<br>**3** From the list of services, select and open the **IIS Admin** service.<br>**4** In the Properties dialog, on the **Log On** tab, select **Local System Account**.<br>**5** Select the option **Allow Service to Interact with Desktop**. |

## Enhanced Error Diagnostics Using mstack Trace

Use this enhanced diagnostic feature to troubleshoot problems that occur specifically during MATLAB code execution.

To implement this feature, use .NET exception handling to invoke the MATLAB function inside of the .NET application, as demonstrated in this try-catch code block:

```
try
{
Magic magic = new Magic();
magic.callmakeerror();
}
catch(Exception ex)
{
Console.WriteLine("Error: {0}", exception);
}
```

When an error occurs, the MATLAB code stack trace is printed before the Microsoft .NET application stack trace, as follows:

```
... MATLAB code Stack Trace ...
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name dmakeerror_error2,line at 14.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name dmakeerror_error1,line at 11.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name dmakeerror,line at 4.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\calldmakeerror.m,name calldmakeerror,line at 2.

... .Application Stack Trace ...
   at MathWorks.MATLAB.NET.Utility.MWMCR.EvaluateFunction
(String functionName, Int32 numArgsOut, Int
32 numArgsIn, MWArray[] argsIn)
   at MathWorks.MATLAB.NET.Utility.MWMCR.EvaluateFunction
(Int32 numArgsOut, String functionName, MWA
rray[] argsIn)
   at CalldmakeerrComp.Calldmakeerr.calldmakeerror() in
h:\compiler\g388611\cathy\MagicDemoComp\src\
Calldmakeerr.cs:line 140
   at MathWorks.Demo.MagicSquareApp.MagicDemoApp.Main(String[]
 args) in H:\compiler\g388611\cathy\Ma
gicDemoCSharpApp\MagicDemoApp.cs:line 52
```

# Reference Information


- “Requirements for the MATLAB® Builder NE Product” on page 11-2
- “Data Conversion Rules” on page 11-4
- “Overview of Data Conversion Classes” on page 11-7
- “MWArray Class Specification” on page 11-14

# Requirements for the MATLAB Builder NE Product

**In this section...**

"System Requirements " on page 11-2
"Compiler Requirements" on page 11-2
"Path Modifications Required for Accessibility" on page 11-2
"Limitations and Restrictions" on page 11-3

## System Requirements

System requirements and restrictions on use for the MATLAB Builder NE product are as follows:

- All requirements for the MATLAB Compiler product; see "Installation and Configuration" in the MATLAB Compiler documentation.
- Microsoft .NET Framework 2.0 or higher must be installed.
- Either Microsoft Visual Studio 2003, Microsoft Visual Studio 2005, or the corresponding .NET Framework SDK must be available on the target machine.

## Compiler Requirements

You must have the MATLAB and MATLAB Compiler products installed to install the MATLAB Builder NE product.

MATLAB Builder NE is available only on Windows (32-bit and 64-bit versions).

For an up-to-date list of all the compilers supported by MATLAB and MATLAB Compiler, see http://www.mathworks.com/support/compilers/current_release/.

## Path Modifications Required for Accessibility

In order to use some screen-readers or assistive technologies, such as JAWS®, you must add the following DLLs to your Windows path:

```
JavaAccessBridge.dll
```

```
WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

## Limitations and Restrictions

In general, limitations and restrictions on the use of the builder are the same as those for MATLAB Compiler. See the MATLAB Compiler documentation for details.

### Using CGI Scripts

As of Release 2006b, CGI scripts can call MATLAB using the Engine API interface if you have a concurrent or designated license.

# Data Conversion Rules

**In this section...**

"Managed Types to MATLAB Arrays" on page 11-4

"MATLAB Arrays to Managed Types" on page 11-5

"Character and String Conversion" on page 11-5

"Unsupported MATLAB Array Types" on page 11-6

## Managed Types to MATLAB Arrays

The following table lists the data conversion rules used when converting native .NET types to MATLAB arrays.

---

**Note** The conversion rules listed in these tables apply to scalars, vectors, matrices, and multidimensional arrays of the native types listed.

---

**Conversion Rules: Managed Types to MATLAB Arrays**

<table>
<tr><th>Native .NET Type</th><th>MATLAB Array</th><th>Comments</th></tr>
<tr><td><code>System.Double</code></td><td><code>double</code></td><td>—</td></tr>
<tr><td><code>System.Single</code></td><td><code>single</code></td><td rowspan="5">Available only when the <code>makeDouble</code> constructor argument is set to <code>false</code>. The default is <code>true</code>, which creates a MATLAB <code>double</code> type.</td></tr>
<tr><td><code>System.Int64</code></td><td><code>int64</code></td></tr>
<tr><td><code>System.Int32</code></td><td><code>int32</code></td></tr>
<tr><td><code>System.Int16</code></td><td><code>int16</code></td></tr>
<tr><td><code>System.Byte</code></td><td><code>int8</code></td></tr>
<tr><td><code>System.String</code></td><td><code>char</code></td><td>None</td></tr>
<tr><td><code>System.Boolean</code></td><td><code>logical</code></td><td>None</td></tr>
</table>

## MATLAB Arrays to Managed Types

The following table lists the data conversion rules used when converting MATLAB arrays to native .NET types.

---

**Note** The conversion rules apply to scalars, vectors, matrices, and multidimensional arrays of the listed MATLAB types.

---

**Conversion Rules: MATLAB Arrays to Managed Types**

| MATLAB Type | .NET Type (Primitive) | .NET Type (Class) | Comments |
|---|---|---|---|
| `cell` | N/A | `MWCellArray` | Cell and struct arrays have no corresponding .NET type. |
| `structure` | N/A | `MWStructArray` | |
| `char` | `System.String` | `MWCharArray` | |
| `double` | `System.Double` | `MWNumericArray` | Default is type double. |
| `single` | `System.Single` | `MWNumericArray` | |
| `uint64` | `System.Int64` | `MWNumericArray` | Not supported |
| `uint32` | `System.Int32` | `MWNumericArray` | Not supported |
| `uint16` | `System.Int16` | `MWNumericArray` | Not supported |
| `uint8` | `System.Byte` | `MWNumericArray` | None |
| `logical` | `System.Boolean` | `MWLogicalArray` | None |
| Function handle | N/A | N/A | None |
| Object | N/A | N/A | None |

## Character and String Conversion

A native .NET string is converted to a `1`-by-*N* MATLAB character array, with *N* equal to the length of the .NET string.

An array of .NET strings (`string[]`) is converted to an *M*-by-*N* character array, with *M* equal to the number of elements in the string (`[]`) array and *N* equal to the maximum string length in the array.

Higher dimensional arrays of `String` are similarly converted.

In general, an *N*-dimensional array of `String` is converted to an *N*+1 dimensional MATLAB character array with appropriate zero padding where supplied strings have different lengths.

## Unsupported MATLAB Array Types

The MATLAB Builder NE product does not support the following MATLAB array types because they are not CLS-compliant:

- `int8`
- `uint16`
- `uint32`
- `uint64`

---

**Note** While it is permissible to pass these types as arguments to a MATLAB Builder NE component, it is not permissible to return these types, as they are not CLS compliant.

---

# Overview of Data Conversion Classes

**In this section...**


"Overview" on page 11-7

"Returning Data from MATLAB to Managed Code" on page 11-8

"Example of MWNumericArray in a .NET Application" on page 11-8

"Interfaces Generated by the MATLAB® Builder NE Product" on page 11-8


## Overview

The data conversion classes are

- `MWArray`
- `MWIndexArray`
- `MWCellArray`
- `MWCharacterArray`
- `MWLogicalArray`
- `MWNumericArray`
- `MWStructArray`

---

**Note** For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

---

`MWArray` and `MWIndexArray` are abstract classes. The other classes represent the standard MATLAB array types: cell, character, logical, numeric, and struct. Each class provides constructors and a set of properties and methods for creating and accessing the state of the underlying MATLAB array.

There are some data types (cell arrays, structure arrays, and arrays of complex numbers) commonly used in the MATLAB product that are not available as native .NET types. To represent these data types, you must create an instance of either`MWCellArray`, `MWStructArray`, or `MWNumericArray`.

## Returning Data from MATLAB to Managed Code

All data returned from a MATLAB function to a .NET method is represented as an instance of the appropriate `MWArray` subclass. For example, a MATLAB cell array is returned as an `MWCellArray` object.

Return data is *not* automatically converted to a native array. If you need to get the corresponding native array type, call the `ToArray` method, which converts a MATLAB array to the appropriate native data type, except for cell and struct arrays. See "Deploying a Component Using the Magic Square Example" on page 1-8.

## Example of MWNumericArray in a .NET Application

Here is a code fragment that shows how to convert a `double` value (`5.0`) to a `MWNumericArray` type:

```
MWNumericArray arraySize = 5.0;
magicSquare = magic.MakeSqr(arraySize);
```

After the `double` value is converted and assigned to the variable `arraySize`, you can use the `arraySize` argument with the MATLAB based method without further conversion. In this example, the MATLAB based method is `magic.MakeSqr(arraySize)`.

## Interfaces Generated by the MATLAB Builder NE Product

For each MATLAB function that you specify as part of a .NET component, the builder generates an API based on the MATLAB function signature, as follows:

- A *single output* signature that assumes that only a single output is required and returns the result in a single `MWArray` rather than an array of `MWArray`s.
- A *standard* signature that specifies inputs of type `MWArray` and returns values as an array of `MWArray`.
- A *feval* signature that includes both input and output arguments in the argument list rather than returning outputs as a return value. Output arguments are specified first, followed by the input arguments.

## Single Output API

**Note** Typically you use the single output interface for MATLAB functions that return a single argument. You can also use the single output interface when you want to use the output of a function as the input to another function.

For each MATLAB function, the builder generates a wrapper class that has overloaded methods to implement the various forms of the generic MATLAB function call. The single output API for a MATLAB function returns a single `MWArray` value.

For example, the following table shows a generic function `foo` along with the single output API that the builder generates for its several forms.

| | |
|---|---|
| Generic MATLAB function | `function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)` |
| API if there are no input arguments | `public MWArray foo()` |
| API if there are one or more input arguments | `public MWArray foo(`<br>`MWArray` *`In1,`*<br>`MWArray` *`In2`*<br>`...`<br>`MWArray` *`inN`*`)` |
| API if there are optional input arguments | `public MWArray foo(`<br>`MWArray` *`In1`*`,`<br>`MWArray` *`In2,`*<br>`...,`<br>`MWArray` *`inN`*<br>`params MWArray[]` *`varargin`*<br>`)` |

In the example, the input arguments *`In1, In2`*, and *`inN`* are of type `MWArray` objects.

Similarly, in the case of optional arguments, the `params` arguments are of type MWArray. (The `varargin` argument is similar to the `varargin` function in MATLAB — it allows the user to pass a variable number of arguments.)

---

**Note** When you call a class method in your .NET application, specify all required inputs first, followed by any optional arguments.

Functions having a single integer input require an explicit cast to type `MWNumericArray` to distinguish the method signature from a standard interface signature that has no input arguments.

---

### Standard API

Typically you use the standard interface for MATLAB functions that return multiple output values.

The standard calling interface returns an array of `MWArray` objects rather than a single array object.

The standard API for a generic function with none, one, more than one, or a variable number of arguments, is shown in the following table.

| | |
|---|---|
| Generic MATLAB function | `function [Out1, Out2, ..., varargout] =`<br>`  foo(In1, In2, ..., InN, varargin)` |
| API if there are no input arguments | `public MWArray[] foo(`<br>`int` *`numArgsOut`*<br>`)` |
| API if there is one input argument | `public MWArray [] foo(`<br>`int` *`numArgsOut`*`,`<br>`MWArray` *`In1`*<br>`)` |

| | |
|---|---|
| API if there are two to *N* input arguments | `public MWArray[] foo(`<br>`int numArgsOut,`<br>`MWArray In1,`<br>`MWArray In2,`<br>`...`<br>`MWArray InN`<br>`)` |
| API if there are optional arguments, represented by the `varargin` argument | `public MWArray[] foo(`<br>`int numArgsOut,`<br>` MWArray in1,`<br>` MWArray in2,`<br>` ...,`<br>`MWArray InN,`<br>`params MWArray[] varargin`<br>`)` |

Details about the arguments for these samples of standard signatures are shown in the following table.

| Argument | Description | Details |
|---|---|---|
| *numArgsOut* | Number of outputs | An integer indicating the number of outputs you want the method to return.<br><br>The value of *numArgsOut* must be less than or equal to the MATLAB function `nargout`.<br><br>The `numArgsOut` argument must always be the first argument in the list. |
| *In1, In2, ...InN* | Required input arguments | All arguments that follow *numArgsOut* in the argument list are inputs to the method being called.<br><br>Specify all required inputs first. Each required input must be of type `MWArray` or one of its derived types. |

| Argument | Description | Details |
|---|---|---|
| *varargin* | Optional inputs | You can also specify optional inputs if your MATLAB code uses the `varargin` input: list the optional inputs, or put them in an `MWArray[]` argument, placing the array last in the argument list. |
| `Out1, Out2, ...OutN` | Output arguments | With the standard calling interface, all output arguments are returned as an array of `MWArray`s. |

### feval API

In addition to the methods in the single API and the standard API, in most cases, the builder produces an additional overloaded method. If the original MATLAB code contains no output arguments, then the builder will not generate the `feval` method interface.

For a function with the following structure,

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

The builder generates the following API, known as the *feval interface*,

```
public void foo
            (int numArgsOut,
             ref MWArray [] ArgsOut,
             MWArray[] ArgsIn)
```

where the arguments are as follows:

| | | |
|---|---|---|
| `numArgsOut` | Number of outputs | Same as standard interface.<br>An integer indicating the number of outputs you want to return.<br>This number generally matches the number of output arguments that follow. The `varargout` array counts as just one argument, if present. |
| `ref MWArray [] ArgsOut` | Output arguments | Following `numArgsOut` are all the outputs of the original MATLAB code, each listed in the same order as they appear on the left side of the original MATLAB code.<br>A `ref` attribute prefaces all output arguments indicating that these arrays are passed by reference. |
| `MWArray[] ArgsIn` | Input arguments | `MWArray` types or a supported .NET primitive type.<br>When you pass an instance of an `MWArray` type, the underlying MATLAB array is passed directly to the called function. Native types are first converted to `MWArray` types. |

# MWArray Class Specification

For complete reference information about the `MWArray` class hierarchy, see the MWArray Class Library Reference (available online only).

See "Making .NET Namespaces Available for Your Generated Component and MWArray Libraries" on page 1-27 for information about referencing the classes in your .NET programming environment.

# 12

# Function Reference

# componentinfo

## Purpose

Query system registry about COM component created with MATLAB Builder NE

## 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

| | |
|---|---|
| *component_name* | MATLAB string naming the COM component created by MATLAB Builder NE. Names are case sensitive. If the argument is not supplied, information is returned on all installed components. |
| *major_revision_number* | Component major revision number. If the argument is not supplied, information is returned on all major revisions. |
| *minor_revision_number* | 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 next.

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

This table describes the fields in componentinfo.

### 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 builder components have the type library bound into the DLL, this file name is the same as the DLL name and path. |

**Registry Information Returned by componentinfo (Continued)**

| Field | Description |
|---|---|
| `Interfaces` | An array of structures defining all interface definitions in the type library. Each structure contains two fields:<br>• `Name` - Interface name.<br>• `IID` - Interface GUID. |
| `CoClasses` | An array of structures defining all COM classes in the component. Each structure contains these fields:<br>• Name - Class name.<br>• CLSID - GUID of the class.<br>• ProgID - Version-dependent program ID.<br>• VerIndProgID - Version-independent program ID.<br>• InprocServer32 - Full name and path to component DLL.<br>• Methods - A structure containing function prototypes of all class methods defined for this interface. This structure contains four fields:<br>- IDL - An array of Interface Description Language function prototypes.<br>- M - An array of MATLAB function prototypes.<br>- C - An array of C-language function prototypes.<br>- VB - An array of VBA function prototypes.<br>• Properties - A cell array containing the names of all class properties.<br>• Events - A structure containing function prototypes of all events defined for this class. This structure contains four fields: |

**Registry Information Returned by componentinfo (Continued)**

| Field | Description |
|---|---|
| | - IDL - An array of Interface Description Language function prototypes.<br>- M - An array of MATLAB function prototypes.<br>- C - An array of C-language function prototypes.<br>- VB - An array of VBA function prototypes. |

## Usage

Use the `componentinfo` function to get information (such as class name, program ID) to pass on to users of a component that you create.

The `componentinfo` function also provides a record of changes made to the registry on your development machine. This information might be useful for debugging if you run into problems.

## Examples

| Function Call | Returned Information |
|---|---|
| `Info = componentinfo` | Information for all installed components. |
| `Info = componentinfo('mycomponent')` | Information for all revisions of `mycomponent`. |
| `Info = componentinfo('mycomponent',1,0)` | Information for revision 1.0 of `mycomponent`. |

# deploytool

**Purpose** Open GUI for MATLAB Builder NE and MATLAB Compiler

**Syntax** `deploytool`

**Description** The `deploytool` command opens the Deployment Tool window, which is the graphical user interface (GUI) for the MATLAB Builder NE and MATLAB Compiler products.

See "Deploying a Component Using the Magic Square Example" on page 1-8 to get started using the Deployment Tool to create .NET and COM components, and see the MATLAB Compiler documentation for information about using the Deployment Tool to create standalone applications and libraries.

See Chapter 1, "Getting Started", for more information about deploying with the GUI.

| Desired Results | Command |
|---|---|
| Start Deployment Tool GUI with the **New/Open** dialog box active | `deploytool` (default)<br>or<br>`deploytool -n` |
| Start Deployment Tool GUI and load *`project_name`* | `deploytool` *`project_name`*`.prj` |
| Start Deployment Tool command line interface and build `project_name` after initializing | `deploytool -win32 -build` *`project_name`*`.prj` |
| Start Deployment Tool command line interface and package `project_name` after initializing | `deploytool -package` *`project_name`*`.prj` |
| Display MATLAB Help for the `deploytool` command | `deploytool -?` |

### -win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are both true:

- You use the same MATLAB installation root (`install_root`) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line (not a MATLAB command line).

# figToImStream

## Purpose

Stream out figure "snapshot" as byte array encoded in format specified, creating signed byte array in .png format

## Syntax

```
output type = figToImStream ('fighandle', figure_handle,
   'imageFormat', image_format, 'outputType', output_type)
```

## Description

The *output type* = figToImStream ('fighandle', *figure_handle*, 'imageFormat', *image_format*, 'outputType', *output_type*) command also accepts user-defined variables for any of the input arguments, passed as a comma-separated list

The size and position of the printed output depends on the figure's PaperPosition[mode] properties.

## Options

figToImStream('figHandle', *Figure_Handle*, ...) allows you to specify the figure output to be used. The Default is the current image

figToImStream('imageFormat', [png|jpg|bmp|gif]) allows you to specify the converted image format. Default value is png.

figToImStream('outputType', [int8!uint8]) allows you to specify an output byte data type. uint8 (unsigned byte) is used primarily for .NET primitive byte. Default value is uint8.

## Examples

Convert the current figure to a signed png byte array:

```
surf(peaks)
bytes = figToImStream
```

Convert a specific figure to an unsigned bmp byte array:

```
f = figure;
surf(peaks);
bytes = figToImStream(  'figHandle', f, ...
                        'imageFormat', 'bmp', ...
                        'outputType', 'uint8' );
```

## Purpose

Invoke MATLAB Compiler

## Syntax

```
mcc -win32 -W 'dotnet:component_name,class_name,
0.0|2.0,Private|Encryption_Key_Path'
file1[file2...fileN]
[class{class_name:file1 [,file2,...,fileN]},...]
[-d output_dir_path]
-T link:lib
```

## Description

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

`mcc` prepares MATLAB file(s) for deployment outside of the MATLAB environment. When used with the MATLAB Builder NE product, wrapper files can be used with all CLS-compliant languages, such as C#, Microsoft Visual Basic .NET, and C++ with Managed Extensions.

For each MATLAB file, the `main` function is a method of the wrapper class generated by MATLAB Builder NE.

## Options

The `-W` option is used when running `mcc` with MATLAB Builder NE.

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

`-W`

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

| -W String Elements | Description |
|---|---|
| `dotnet:` | Keyword that tells the compiler the type of component to create, followed by a colon. Specify `dotnet` to create a .NET component. |
| *`component_name`* | Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`. |
| *`class_name`* | Specifies the name of the .NET class to be created. |
| `0.0\|2.0` | Specifies the version of the .NET Framework you want to use to compile the component. You can specify either:<br>`0.0` — Use the latest supported version on the target machine.<br>`2.0` — Use Version 2.0 of the framework. |
| `Private\|Encryption_Key_Path` | Specifies whether the component to be created is a private assembly or a shared assembly. To create a shared assembly, you must specify the full path to the encryption key file used to sign the assembly. |
| `local\|remote` | Specifies the remoting type of the component. See Chapter 9, "Sharing Components Across Distributed Applications Using .NET Remoting". |

*`file1`* `[`*`file2...fileN`*`]`

Specifies the MATLAB files that are to be encapsulated as methods in the class being created (*`class_name`*).

`class{`*`class_name`*`:`*`file1`* `[,`*`file2`*`,...,`*`fileN`*`]},...`

(Optional) Specifies additional classes that you want to include in the component. To use this option, you specify the class name, followed by a colon, and then the names of the files you want to include in the class. You can include this multiple times to specify multiple classes.

[-d *output_dir_path*]

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

`-C`

Overrides automatically embedding the CTF archive in builder-generated .NET or COM components. See for details.

`-T`

Specifies the output type. To create a .NET component, specify the keyword `link:lib`, which links objects into a shared library (DLL).

### -win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are both true:

- You use the same MATLAB installation root (*install_root*) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line (not a MATLAB command line).

13

# Creating and Installing COM Components


- "Building a Deployable COM Component" on page 13-2
- "Packaging a Deployable COM Component" on page 13-4
- "About Embedded CTF Archives" on page 13-5
- "Using the Command-Line Interface" on page 13-6
- "Installing COM Components on a Target Computer" on page 13-9

# Building a Deployable COM Component

1. Create a deployment project. A *project* is a collection of files you bundle together under a project file name (`.prj` file) for your convenience in the Deployment Tool. Using a project makes it easy for you to build and run an application many times—effectively testing it—until it is ready for deployment.

   a. Type the name of your project in the **Name** field.

   b. Enter the location of the project in the **Location** field. Alternately, navigate to the location.

   c. Select the target for the deployment project from the **Target** drop-down menu.

   d. Click **OK**.

2. On the **Build** tab, add what you want to compile, and any supporting files, to the project.

   a. Do the following, depending on the type of application you are building:

      - If you are building a COM application or Microsoft Excel add-in, click **Add files**.

      - If you are building a .NET application, click **Add class**. Type the name of the class in the Class Name field, designated by the letter “c”: For this class, add files you want to compile by clicking **Add files**. To add another class, click **Add class**.

   b. Add any supporting files. For example, you can add the following files, as appropriate for your project:

      - Functions called using `eval` (or variants of `eval`)
      - Functions not on the MATLAB path
      - Code you want to remain private
      - Code from other programs that you want to compile and link into the main file

   If you want to include additional files, in the Shared Resources and Helper Files area, click **Add files/directories**. Click **Open** to select the file or files.

**3** When you complete your changes, click the Build button.

# Packaging a Deployable COM Component

1. On the **Package** tab, add the MATLAB Compiler Runtime (the MCR) by clicking **Add MCR**.

2. Next, add others files useful for end users. The `readme.txt` file contains important information about others files useful for end users. To package additional files or folders, click **Add file/directories**, select the file or folder you want to package, and click **Open**.

3. In the Deployment Tool, click the Packaging button.

4. After packaging, the package resides in the `distrib` subfolder. On Windows, the package is a self-extracting executable. On platforms other than Windows, it is a `.zip` file. Verify that the contents of the `distrib` folder contains the files you specified.

# About Embedded CTF Archives

As of R2008b, the MATLAB Builder NE product now embeds the CTF archive within generated components, by default. This offers convenient deployment of a single output file since all encrypted MATLAB file data is now contained within the component.

For information on how to produce a separate CTF archive (the default behavior before R2008b), see “Extracting the CTF Archive Manually Using the MCR Component Cache” on page 4-32.

# Using the Command-Line Interface

A MATLAB class cannot be directly compiled into a COM object. You can, however, use a user-generated class inside an MATLAB file and build a COM object from that file. You can use the MATLAB command-line interface instead of the GUI to create COM objects. Do this by issuing the `mcc` command with options. If you use `mcc`, you do not create a project.

---

**Note** See the MATLAB Compiler documentation for a complete description of the `mcc` command and its options.

---

The following table provides an overview of some `mcc` options related to components, along with syntax and examples of their usage.

### Using the Command Line to Create COM Components

<table>
<tr><th>Action to Perform</th><th>mcc Option to Use</th><th>Description</th></tr>
<tr><td rowspan="3">Create component that has one class.</td><td>-W com</td><td>The W option with com as the type controls the generation of wrapper files, which you can use to support components.</td></tr>
<tr><td colspan="2">Syntax<br>mcc -W<br>'com:<component_name>[,<class_name>[,<major>.<minor>]]'<br><br>An unspecified <class_name> defaults to <component_name>, and an unspecified version number defaults to the latest version built or 1.0, if there is no previous version.</td></tr>
<tr><td colspan="2">Example<br>mcc -W 'com:mycomponent,myclass,1.0' -T link:lib foo.m bar.m<br><br>The example creates a COM component called mycomponent, which contains a single COM class named myclass with methods foo and bar, and a version of 1.0.</td></tr>
</table>

**Using the Command Line to Create COM Components (Continued)**

| Action to Perform | mcc Option to Use | Description |
|---|---|---|
| Add additional classes to a COM component. | Not needed | A separate COM named `<class_name>` is created for each class argument that is passed.<br><br>Following the `<class_name>` parameter is a comma-separated list of source files that are encapsulated as methods for the class. |
| | **Syntax**<br>`class{<class_name>:[file, [file,...]]}` | |
| | **Example**<br>`mcc -B 'ccom:mycomponent,myclass,1.0'`<br>`foo.m bar.m class{myclass2:foo2.m, bar2.m}`<br><br>The example creates a COM component named `mycomponent` with two classes: `myclass` has methods `foo` and `bar`, and `myclass2` has methods `foo2` and `bar2`. The version is version 1.0. | |
| Simplify the command-line input for components. | `-B ccom:` | Uses the bundle file. |
| | **Syntax**<br>`mcc -B '<filename>'[:<a1>,<a2>,...,<an>]` | |
| | **Example**<br>`mcc -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m` | |
| Control how each COM class uses the MCR. | `-S` | By default, a new MCR instance is created for each instance of each COM class in the component. Use `-S` to change the default.<br><br>This option tells the builder to create a single MCR at the time when the first COM class is instantiated. This MCR is reused and shared among all subsequent class instances, resulting in more efficient memory usage and eliminating the MCR startup cost in each subsequent class instantiation.<br>When using `-S`, note that all class instances share a single MATLAB workspace and share global variables in the MATLAB files used to build the component. Therefore, properties of a |

**Using the Command Line to Create COM Components (Continued)**

<table>
<tr><th>Action to Perform</th><th>mcc Option to Use</th><th>Description</th></tr>
<tr><td rowspan="2"></td><td></td><td>COM class behave as static properties instead of instance-wise properties.<br><br>Note The default behavior dictates that a new MCR be created for each instance of a class, so when the class is destroyed, the MCR is destroyed as well. If you want to retain the state of global variables (such as those allocated for drawing figures, for instance), use the <code>-S</code> option.</td></tr>
<tr><td colspan="2">Example<br><code>mcc -S -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m</code><br><br>The example creates a COM component called <code>mycomponent</code> containing a single COM class named <code>myclass</code> with methods <code>foo</code> and <code>bar</code>, and a version of 1.0.<br><br>When multiple instances of this class are instantiated in an application, only one MCR is initialized, and it is shared by each instance.</td></tr>
<tr><td rowspan="2">Create subfolders needed for deployment and copy associated files to them.</td><td><code>-d</code></td><td>The <code>\src</code> and <code>\distrib</code> subfolders are needed to package components.</td></tr>
<tr><td colspan="2">Syntax<br><code>-d foldername</code></td></tr>
</table>

# Installing COM Components on a Target Computer

To install and deploy a COM object created with MATLAB Builder NE, perform the following steps:

1. Install the MATLAB Compiler Runtime as described in the *MATLAB Compiler User's Guide*.

2. Build and package as described in "Building a Deployable COM Component" on page 13-2 and "Packaging a Deployable COM Component" on page 13-4.

3. Copy the package to the target computer and run the package.

4. From a Windows command prompt on the target system, navigate to the folder where you saved the package. If you use the command `dir`, you should see the `.dll` created for your COM object. You will need to register the `.dll` manually using the command `regsvr32`, as follows:

   ```
   regsvr32 myCom_1_0.dll
   ```

14

# Programming with COM Components Created by the MATLAB Builder NE Product


- "General Techniques" on page 14-2
- "Registering and Referencing the Utility Library" on page 14-4
- "Creating an Instance of a Class in Microsoft® Visual Basic" on page 14-5
- "Calling the Methods of a Class Instance" on page 14-8
- "Calling a COM Object in a Visual C++ Program" on page 14-11
- "Using a COM Component in a .NET Application" on page 14-14
- "Adding Events to COM Objects" on page 14-21
- "Passing Arguments " on page 14-26
- "Using Flags to Control Array Formatting and Data Conversion" on page 14-29
- "Using MATLAB Global Variables in Microsoft® Visual Basic" on page 14-36
- "Blocking Execution of a Console Application that Creates Figures" on page 14-39
- "Obtaining Registry Information" on page 14-42
- "Handling Errors During a Method Call" on page 14-44

# General Techniques

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

Your code module must do the following:

- Load the components created by the builder
  - "Registering and Referencing the Utility Library" on page 14-4
  - "Creating an Instance of a Class in Microsoft® Visual Basic" on page 14-5
- Call methods of the component class
  - "Calling the Methods of a Class Instance" on page 14-8
  - "Calling a COM Object in a Visual C++ Program" on page 14-11
  - "Adding Events to COM Objects" on page 14-21
  - "Obtaining Registry Information" on page 14-42
- Deal with data conversion and parameter passing
  - "Passing Arguments " on page 14-26
  - "Using Flags to Control Array Formatting and Data Conversion" on page 14-29
  - "Using MATLAB Global Variables in Microsoft® Visual Basic" on page 14-36
- Process errors
  - "Handling Errors During a Method Call" on page 14-44

---

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

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

---

# Registering and Referencing the Utility Library

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

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

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

```
mwregsvr mwcomutil.dll
```

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

1 Select **Tools > References**.

2 Select **MWComUtil 7.5 Type Library**.

---

**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.

---

# Creating an Instance of a Class in Microsoft Visual Basic

**In this section...**

"Advantages and Disadvantages" on page 14-5

"CreateObject Function" on page 14-5

"Microsoft® Visual Basic New Operator" on page 14-6

"Advantages of Each Technique" on page 14-7

"Declaring a Reusable Class Instance" on page 14-7

## Advantages and Disadvantages

Each technique listed here has advantages and disadvantages.

For an example of creating a class instance in Microsoft Visual C++, see "Calling a COM Object in a Visual C++ Program" on page 14-11.

## CreateObject Function

This method uses the Microsoft Visual Basic application program interface (API) `CreateObject` function to create an instance of the class.

1. Dimension a variable of type `Object` to hold a reference to the class instance.

2. Call `CreateObject` with the Program ID (`ProgID`) for the class as an argument.

Here is a programming example:

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

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

```
    foo = Err.Description
End Function
```

## Microsoft Visual Basic New Operator

This method uses the Microsoft Visual Basic `New` operator on a variable explicitly dimensioned as the class to be created.

1. Make sure that you reference the type library containing the class in the current Visual Basic project.
   a. Open the Visual Basic editor.
   b. Select **Project > References > Available References**.
   c. Select the necessary type library.
2. Dimension the class instance.
3. Use `New` to instantiate the class with a particular name.

The following sample function, `foo`, shows how to use the `New` operator to create a class instance:

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

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

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

## Advantages of Each Technique

Both techniques (using `CreateObject` and using `New`) are equivalent in the way they function, but each has different advantages. The first technique does not require a reference to the type library in the Visual Basic project, while the second results in faster code execution. The second technique has the added advantage of enabling **Auto-List-Members** and **Auto-Quick-Info** in the Visual Basic editor to help you work with your classes.

## Declaring a Reusable Class Instance

In the previous examples, the class instance used to call the method is a local variable within a procedure. Thus a new class instance is created and destroyed for each call to the method. As an alternative, you can declare a single module-scoped class instance that is reused by all function calls. The next example shows this technique:

```
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
```

# Calling the Methods of a Class Instance

**In this section...**


"Standard Mapping Technique" on page 14-8

"Variant" on page 14-9

"Examples of Passing Input and Output Parameters" on page 14-9


## Standard Mapping Technique

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

- `nargout`

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

- Output parameters

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

- Input parameters

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

For example, the most generic MATLAB function is:

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

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

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

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

See "Calling Conventions" on page 16-23 for more details and examples of the standard mapping from MATLAB functions to COM class method calls.

## Variant

All input and output arguments are typed as `Variant`, the default Visual Basic data type. The `Variant` type can hold any of the basic Visual Basic types, arrays of any type, and object references. See "Data Conversion" on page 16-9 for details about the conversion of any basic type to and from MATLAB data types.

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

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

## Examples of Passing Input and Output Parameters

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

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

```
Function foo(x1 As Variant, x2 As Variant) As Variant
```

```
    Dim aClass As Object
    Dim y As Variant

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

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

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

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

# Calling a COM Object in a Visual C++ Program

**In this section...**

"Using the MATLAB® Builder NE Product to Create the Object" on page 14-11

"Using the Component in a Visual C++ Program" on page 14-12

---

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

---

## Using the MATLAB Builder NE Product to Create the Object

Build the COM object as follows:

1. Start the MATLAB product.

2. Open the MATLAB Editor and create a file named `adddoubles.m` with the following MATLAB code:

```
function z=adddoubles(x,y)
z=x+y;
```

3. In the MATLAB Command Window, issue the following command to open the Deployment Tool:

```
deploytool
```

4. Create a project named `mycomponent` in any location you want.

5. Add `adddoubles.m` to the `mycomponentclass` folder. This means that the MATLAB function, `adddoubles`, will be a method in `mycomponentclass`.

6. Click the icon in the Deployment Tool toolbar.

   The builder generates a self-registering COM object that you can use in your Visual C++® code.

## Using the Component in a Visual C++ Program

Use the COM object you have created as follows:

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

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

// include the following files generated by MATLAB Builder NE
#include "mycomponent\src\mycomponent_idl.h"
#include "mycomponent\src\mycomponent_idl_i.c"

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

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

```
mbuild matlab_com_example.cpp
```

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

# Using a COM Component in a .NET Application

**In this section...**

"Overview" on page 14-14

"C# Implementation" on page 14-14

"Microsoft® Visual Basic Implementation" on page 14-17

## Overview

The following examples demonstrate the optimal fitting of a nonlinear function to a set of data in both C# and Microsoft Visual Basic implementations.

Note in particular how memory is freed and allocated. Use these examples as models when using COM components in your own .NET applications.

## C# Implementation

```
// *****************************************************************************
//
// CurveFitApp.cs
//
// This file is an example for using MATLAB COM component inside .NET application.
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *****************************************************************************

using System;
using CurveFitDemoComp;


namespace MathWorks.Demo.CurveFitApp
{
        /// args[0] - a positive integer representing number of
        ///     data points.
        ///     class CurveFitApp
    {
```

```
///         /// The main entry point for the application.
///         [STAThread]
static void Main(string[] args)
{
    CurveFitClass curveFitting = null;
    try
    {
        // Get user specified command line arguments or
        // set default
        int numberOfDataPts= (0 != args.Length)
              ? System.Int32.Parse(args[0]) : 4;

        // Input that will be passed to the COM method
        double[] xData = new double[numberOfDataPts];
        double[] yData = new double[numberOfDataPts];
        for(int i=1; i<= numberOfDataPts; i++)
        {
            xData[i-1] = i;
            yData[i-1] = i;
        }

        // Objects that will be returned by the COM method
        object coefficients = new object();
        object lambda = new object();

        // Create the curve fit object
        curveFitting = new CurveFitClass();
        if(curveFitting != null)
        {
            curveFitting.MWFlags.ArrayFormatFlags.TransposeOutput =
                                                                  true;

            // Call the COM method
            curveFitting.fitdemo(2, ref coefficients,
                            ref lambda, xData, yData);

            // Display values of co-efficients returned by COM method
            Console.WriteLine("\nCo-efficient:\n");

            // Convert the coefficients array to a two
```

```
            // dimensional native double array
            if(coefficients.GetType().IsArray)
            {
                System.Array coeffArray =
                (System.Array)coefficients;
                // Display the array elements:
                for (int i= coeffArray.GetLowerBound(0);
                     i <= (int)coeffArray.GetUpperBound(0); i++)
                   for (int j= coeffArray.GetLowerBound(1);
                     j <= (int)coeffArray.GetUpperBound(1); j++)
                       Console.WriteLine("Result({0},{1})= {2}", i, j,
                       coeffArray.GetValue(i,j));
            }

            // Display values of lambda returned by COM method
            Console.WriteLine("\nLambda:\n");

            // Convert the lambda array to a two dimensional
            // native double array
            if(lambda.GetType().IsArray)
            {
                System.Array lambdaArray = (System.Array)lambda;
                // Display the array elements:
                for (int i= lambdaArray.GetLowerBound(0);
                    i <= (int)lambdaArray.GetUpperBound(0); i++)
                  for (int j= lambdaArray.GetLowerBound(1);
                    j <= (int)lambdaArray.GetUpperBound(1); j++)
                      Console.WriteLine("Result({0},{1})= {2}", i, j,
                      lambdaArray.GetValue(i,j));
            }

        }
        Console.ReadLine();  // Wait for user to exit application

    }
    catch(System.Runtime.InteropServices.COMException exception)
    {
        Console.WriteLine("COM Error: {0}", exception);
    }
    catch(Exception exception)
```

```
            {
                Console.WriteLine("Error: {0}", exception);
            }
            finally
            {
                // Free COM object
                if(curveFitting != null)
                    System.Runtime.InteropServices.Marshal.ReleaseComObject(curveFitting);
            }
        }
    }
}
```

## Microsoft Visual Basic Implementation

```
' ********************************************************************************
'
' CurveFitApp.vb
'
' This file is an example for using MATLAB COM component inside .NET Visaul
' Basic application.
'
' Copyright 2001-2010 The MathWorks, Inc.
'
' ********************************************************************************

Imports System
Imports CurveFitDemoComp

Namespace MathWorks.Demo.CurveFitApp


    ' <remarks>
    ' args[0] - a positive integer representing number of data points.
    ' </remarks>
         Public Class CurveFitApp
#Region " MAIN "

    ' <summary>
```

```
' The main entry point for the application.
' </summary>
        Shared Sub Main(ByVal args() As String)
        Dim curveFitting As CurveFitClass = Nothing

        Try
            ' Get user specified command line arguments or set default
            Dim numberOfDataPts As Integer
            If (0 <> args.Length) Then
                numberOfDataPts = System.Int32.Parse(args(0))
            Else
                numberOfDataPts = 4
            End If

            ' Input that will be passed to the COM method
            Dim xData() As Double = New Double(numberOfDataPts - 1) {}
            Dim yData() As Double = New Double(numberOfDataPts - 1) {}

            For i As Integer = 1 To numberOfDataPts
                xData(i - 1) = i
                yData(i - 1) = i
            Next i

            ' Objects that will be returned by the COM method
            Dim coefficients As Object = New Object
            Dim lambda As Object = New Object

            ' Create the curve fit object
            curveFitting = New CurveFitClass
            If Not (curveFitting Is Nothing) Then

                curveFitting.MWFlags.ArrayFormatFlags.TransposeOutput = True

                ' Call the COM method
                curveFitting.fitdemo(2, coefficients, lambda, xData, yData)

                ' Display values of co-efficients returned by COM method
                Console.WriteLine("{0}Co-efficient:{1}", Chr(10), Chr(10))

                ' Convert the coefficients array to a two dimensional
```

```
        ' native double array
        If (coefficients.GetType().IsArray) Then

            Dim coeffArray As System.Array = CType(coefficients,
                                                System.Array)

            ' Display the array elements:
            For i As Integer = coeffArray.GetLowerBound(0)
               To coeffArray.GetUpperBound(0)
                 For j As Integer = coeffArray.GetLowerBound(1)
                   To coeffArray.GetUpperBound(1)
                    Console.WriteLine("Result({0},{1})= {2}", i, j,
                    coeffArray.GetValue(i, j))
                 Next j
            Next i
        End If

        ' Display values of lambda returned by COM method
        Console.WriteLine("{0}Lambda:{1}", Chr(10), Chr(10))


        ' Convert the lambda array to a two dimensional native
        ' double array
        If (lambda.GetType().IsArray) Then
            Dim lambdaArray As System.Array = CType(lambda,
                                                System.Array)
            ' Display the array elements:
            For i As Integer = lambdaArray.GetLowerBound(0)
              To lambdaArray.GetUpperBound(0)
                For j As Integer = lambdaArray.GetLowerBound(1
                ) To lambdaArray.GetUpperBound(1)
                    Console.WriteLine("Result({0},{1})= {2}", i, j,
                    lambdaArray.GetValue(i, j))
                Next j
            Next i
        End If

    End If
    Console.ReadLine()  ' Wait for user to exit application
```

```
            Catch exception As System.Runtime.InteropServices.COMException
                Console.WriteLine("COM Error: {0}", exception)

            Catch exception As Exception
                Console.WriteLine("Error: {0}", exception)

            Finally
                ' Free COM object
                If Not (curveFitting Is Nothing) Then
                    System.Runtime.InteropServices.Marshal.ReleaseComObject(curveFitting)
                End If
            End Try

        End Sub

#End Region
    End Class
End Namespace
```

# Adding Events to COM Objects

**In this section...**

"MATLAB Language Pragma" on page 14-21

"Using a Callback with a Microsoft® Visual Basic Event" on page 14-22

## MATLAB Language Pragma

The MATLAB Builder NE product supports events, or callbacks, through a MATLAB language pragma. A *pragma* is a directive to the builder, beyond what is conveyed in the MATLAB language itself. The pragma for adding events is `#event`.

The MATLAB product interprets the `%#event` statement as a comment. But when the builder encapsulates a function, the `#event` pragma tells the builder that the function requires an *outgoing interface* and an *event handler*.

---

**Note** The `#event` pragma is supported only for COM components built with MATLAB Builder NE. You can not use this feature with .NET components created by MATLAB Builder NE or COM components built with the MATLAB Builder EX product.

---

To use the `#event` pragma:

1. Write the code for a MATLAB function stub that serves as the prototype for the event. This function stub is the *event function*.

2. Build the COM component as usual. Make sure that you specify the event function you wrote in the MATLAB product as a method in the component class.

3. In your application, add the code to implement the event handler (the event handler belongs to the COM object created by the builder). The code for the event handler should implement the event function, or function stub, that you wrote in MATLAB.

When an encapsulated MATLAB function (now a method in a COM object in your application) calls the event function, the call is dispatched to the event handler in the application.

Some examples of how you might use callbacks in your code are

- To give the application periodic feedback during a long-running calculation by an encapsulated MATLAB function. For example, if you have a task that requires n iterations, you might signal an event to increment a progress bar in the user interface on each iteration.
- To signal a warning during a calculation but continue execution of the task.
- To return intermediate results of a calculation to the user and continue execution of the task.

## Using a Callback with a Microsoft Visual Basic Event

The example in this topic shows how to use a callback in conjunction with a Microsoft Visual Basic `ProgressBar` control.

The MATLAB function `iterate` runs through `n` iterations and fires an event every `inc` iterations. When the function finishes, it returns a single output. To simulate actually doing something, the sample code includes a `pause` statement in the main loop so that the function waits for 1 second in each iteration.

The sample includes MATLAB functions `iterate.m` and `progress.m`.

### iterate.m

```
function [x] = iterate(n,inc)
    %initialize x
    x = 0;
    % Run n iterations, callback every inc time
    k = 0;
    for i=1:n
        k = k + 1;
        if k == inc
            progress(i);
            k = 0;
        end;
```

```
        % Do some work on x...
        x = x + 1;
        % Pause for 1 second to simulate doing
        % something
        pause(1);
    end;
```

**progess.m**

```
function progress(i)
    %#event
    i
```

The `iterate` function runs through `n` iterations and calls the `progress` function every `inc` iterations, passing the current iteration number as an argument. When this function is executed in MATLAB, the value of `i` appears each time the `progress` function gets called.

Suppose you create a the builder component that has these two functions included as class methods. For this example the component has a single class named `myclass`. The resulting COM class has a method `iterate` and an event `progress`.

To receive the event calls, implement a "listener" in the application. The Visual Basic syntax for the event handler for this example is

```
Sub aClass_progress(ByVal i As Variant)
```

where `aClass` is the variable name used for your class instance. The `ByVal` qualifier is used on all input parameters of an event function. To enable the listening process, dimension the `aClass` variable with the `WithEvents` keyword.

This example uses a simple Visual Basic form with three `TextBox` controls, one `CommandButton` control, and one `ProgressBar` control. The first text box, `Text1`, inputs the number of iterations, stored in the form variable `N`. The second text box, `Text2`, inputs the callback increment, stored in the variable `Inc`. The third text box, `Text3`, displays the output of the function when it finishes executing. The command button, `Command1`, executes the `iterate`

method on your class when pressed. The progress bar control, ProgressBar1, updates itself in response to the progress event.

```
'Form Variables
Private WithEvents aClass As myclass    'Class instance
Private N As Long                       'Number of iterations
Private Inc As Long                     'Callback increment
Private Sub Form_Load()
'When form is loaded, create new myclass instance
    Set aClass = New myclass
    'Initialize variables
    N = 2
    Inc = 1
End Sub
Private Sub Text1_Change()
'Update value of N from Text1 text whenever it changes
    On Error Resume Next
    N = CLng(Text1.Text)
    If Err <> 0 Then N = 2
    If N < 2 Then N = 2
End Sub
Private Sub Text2_Change()
'Update value of Inc from Text2 text whenever it changes
    On Error Resume Next
    Inc = CLng(Text2.Text)
    If Err <> 0 Then Inc = 1
    If Inc <= 0 Then Inc = 1
End Sub
Private Sub Command1_Click()
'Execute function whenever Execute button is clicked
    Dim x As Variant
    On Error GoTo Handle_Error
    'Initialize ProgressBar
    ProgressBar1.Min = 1
    ProgressBar1.Max = N
    Text3.Text = ""
    'Iterate N times and call back at Inc intervals
    Call aClass.iterate(1, x, CDbl(N), CDbl(Inc))
    Text3.Text = Format(x)
    Exit Sub
```

```
Handle_Error:
    MsgBox (Err.Description)
End Sub
Private Sub aClass_progress(ByVal i As Variant)
'Event handler. Called each time the iterate function
'calls the progress function. Progress bar is updated
'with the value passed in, causing the control to advance.
    ProgressBar1.Value = i
End Sub
```

# Passing Arguments

**In this section...**


"Overview" on page 14-26

"Creating and Using a varargin Array in Microsoft® Visual Basic Programs" on page 14-26

"Creating and Using varargout in Microsoft® Visual Basic Programs" on page 14-27

"Passing an Empty varargin From Microsoft® Visual Basic Code" on page 14-28


## Overview

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

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

See "Producing a COM Class" on page 16-23 for more information about mapping of input and output arguments.

## Creating and Using a varargin Array in Microsoft Visual Basic Programs

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

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

```
Function foo(x1 As Variant, x2 As Variant, x3 As Variant, _
             x4 As Variant, x5 As Variant) As Variant
```

```
   Dim aClass As Object
   Dim v(1 To 5) As Variant
   Dim y As Variant

   On Error Goto Handle_Error
   v(1) = x1
   v(2) = x2
   v(3) = x3
   v(4) = x4
   v(5) = x5
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,y,v)
   foo = y
   Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

## Creating and Using varargout in Microsoft Visual Basic Programs

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

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

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

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

```
End Sub
```

## 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.

### Example: 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
```

# Using Flags to Control Array Formatting and Data Conversion

**In this section...**

"Overview" on page 14-29
"Array Formatting Flags" on page 14-30
"Using Array Formatting Flags" on page 14-30
"Using Data Conversion Flags" on page 14-33
"Special Flags for Some Microsoft® Visual Basic Types" on page 14-35

## Overview

Generally, you should write your application code so that it matches the arguments (input and output) of the MATLAB functions that are encapsulated in the COM objects that you are using. The mapping of arguments from the MATLAB product to Microsoft Visual Basic is fully described in MATLAB® to COM VARIANT Conversion Rules on page 16-12 and COM VARIANT to MATLAB® Conversion Rules on page 16-17.

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

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

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.

## Array Formatting Flags

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

| Name of Flag | Possible Values of Flag | Results of Conversion |
|---|---|---|
| `InputArrayFormat` | `mwArrayFormatMatrix` (default) | MATLAB matrix from general `Variant` data. |
| | `mwArrayFormatCell` | MATLAB cell array from general `Variant` data. |
| | Array data from an Excel `range` is coded in Visual Basic as an array of `Variant`. Since MATLAB functions typically have matrix arguments, using the default setting makes sense when you are dealing with data from Excel. | |
| `OutputArrayFormat` | `mwArrayFormatAsIs` | Array of `Variant` |
| | Converts arrays according to the default conversion rules listed in MATLAB® to COM VARIANT Conversion Rules on page 16-12. | |
| | `mwArrayFormatMatrix` | A `Variant` containing an array of a basic type. |
| | `mwArrayFormatCell` | MATLAB cell array from general `Variant` data. |
| `AutoResizeOutput` | When this flag is set, the target range automatically resizes to fit the resulting array. If this flag is not set, the target range must be at least as large as the output array or the data is truncated. Use this flag for Excel `Range` objects passed directly as output parameters. | |
| `TransposeOutput` | Transposes all array output. Use this flag when dealing with an encapsulated MATLAB function whose output is a one-dimensional array. By default, the MATLAB product handles one-dimensional arrays as 1-by-n matrices (that is, as row vectors). Change this default with the `TransposeOutput` flag if you prefer column output. | |

## Using Array Formatting Flags

To use the following example, make sure that you reference the `MWComUtil` library in the current project:

1 Select **Tools > References**.

2 Click **MWComUtil 7.5 Type Library**.

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

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

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

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

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

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

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

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

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

### Modifying Output Format

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

### Output Format in VBScript

When calling a COM object in VBScript you need to make sure that you set `MWFlags` for the COM object to specify cell array for the output. Also, you

must use an enumeration (the enumeration value for a cell array is 2) to make the specification (rather than specifying mwArrayFormatCell).

The following sample code shows how to accomplish this:

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

## Using Data Conversion Flags

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

To use the following example, make sure that you reference the MWComUtil library in the current project:

1. Select **Tools > References**.

2. Click **MWComUtil 7.5 Type Library**.

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

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

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

If the original MATLAB function expects 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. As an alternative, you can set the

CoerceNumericToType flag to mwTypeDouble, causing the data converter to convert all numeric input to double. To do this, place the following line after creating the class and before calling the methods:

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

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

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

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

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

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

 On Error Goto Handle_Error
 today = Now
 Set aClass = New mycomponent.myclass
 Set ytemp = New MWArg
```

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

## Special Flags for Some Microsoft Visual Basic Types

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

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

---

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

---

# Using MATLAB Global Variables in Microsoft Visual Basic

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

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

Properties are particularly useful when you have a large array containing values that do not change often, but you need to operate on it frequently. In this case, you can set the array once as a class property and operate on it repeatedly without incurring the overhead of passing (and converting) the data for passing to each method every time it is called.

The following example shows how to use a class property in a matrix factorization class. The example develops a class that performs Cholesky, LU, and QR factorizations on the same matrix. It stores the input matrix (coded as `A` in MATLAB) as a class property so that it does not need to be passed to the factorization routines.

Consider these three MATLAB files.

`Cholesky.m`

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

`LUDecomp.m`

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

```
        return;
    end
    [L,U] = lu(A);
```

QRDecomp.m

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

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

To build the class:

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

To test your application, make sure that you reference the library generated by the builder in the current Visual Basic project:

1. Select **Project > References** in the Visual Basic main menu.
2. Click **mymatrix 1.0 Type Library**.

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

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

```
        Q As Variant, R As Variant
        Dim factor As myfactor

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

Run the subroutine, which does the following:

1. Creates an instance of the `myfactor` class
2. Assigns a double matrix to the property `A`
3. Calls the three factorization methods

# Blocking Execution of a Console Application that Creates Figures

**In this section...**

"MCRWaitForFigures" on page 14-39

"Using MCRWaitForFigures to Block Execution" on page 14-40

## MCRWaitForFigures

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

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

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

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

---

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

---

## Using MCRWaitForFigures to Block Execution

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

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

2 Create the following MATLAB file in this folder:

```
drawplot.m

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

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

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

---

**Note** Instead of using the Deployment Tool, you can create the component by issuing the following command at the MATLAB prompt:

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

---

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

```
// Include the following files generated by
// MATLAB Builder NE:
#include "src\plotdemo_idl.h"
#include "src\plotdemo_idl_i.c"

int main()
```

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

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

```
mbuild runplot.cpp
```

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

---

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

---

# Obtaining Registry Information

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

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

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

Info =

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

Info.Interfaces

ans =

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

Info.CoClasses

ans =

          Name: 'myclass'
         CLSID: '{3A14AB35-44BE-11D5-B155-00D0B7BA7544}'
        ProgID: 'mycomponent.myclass.1_0'
   VerIndProgID: 'mycomponent.myclass'
InprocServer32:'D:\Work\mycomponent\distrib\mycomponent_1_0.dll'
```

```
       Methods: [1x4 struct]
    Properties: {'m', 'n'}
        Events: [1x1 struct]

Info.CoClasses.Events.M

ans =

function myevent(x, y)

Info.CoClasses.Methods

ans =

1x4 struct array with fields:
    IDL
    M
    C
    VB

Info.CoClasses.Methods.M

ans =

function [y] = mysum(varargin)

ans =

function [varargout] = randvectors()

ans =

function [x] = getdates(n, inc)

ans =

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

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

## Handling Errors During a Method Call

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

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

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

# 15

# Using COM Components in Microsoft Visual Basic Applications


- “Magic Square Example” on page 15-2
- “Creating an Excel Add-In: Spectral Analysis Example” on page 15-10
- “Univariate Interpolation Example” on page 15-25
- “Matrix Calculator Example” on page 15-33
- “Curve Fitting Example” on page 15-44
- “Bouncing Ball Simulation Example” on page 15-52

# Magic Square Example

**In this section...**


"Example Overview" on page 15-2

"Creating the MATLAB File" on page 15-2

"Using the Deployment Tool to Create and Build the Project" on page 15-3

"Creating the Microsoft® Visual Basic Project" on page 15-3

"Creating the User Interface" on page 15-4

"Creating the Executable in Microsoft® Visual Basic" on page 15-7

"Testing the Application" on page 15-7

"Packaging the Component" on page 15-8


## Example Overview

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

---

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

---

## Creating the MATLAB File

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

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

## Using the Deployment Tool to Create and Build the Project

**1** Specify a COM component as follows:

**a** While in MATLAB, issue the following command to open Deployment Tool:

```
deploytool
```

**b** Create a project with the following settings:

| Setting | Value |
|---|---|
| Project name | `magicdemo` |
| Class name | `magicdemoclass` |
| Project folder | The name of your work folder followed by the component name. In this example, that is `D:\Work\MagicSquareExample\magicdemo`. |
| Generate Verbose Output | Selected |

**c** Locate your work folder and navigate to the `MagicDemoComp` folder, which contains the MATLAB file for the `makesquare` function. Add the `makesquare.m` file to the project.

**2** Build the component by clicking the button in the Deployment Tool toolbar.

The build process begins, and a log of the build appears in the **Output** pane of the Deployment Tool. The files that are needed for the component are copied to two newly created folders, `src` and `distrib`, in the `magicdemo` folder. A copy of the build log is placed in the `src` folder.

## Creating the Microsoft Visual Basic Project

---

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

---

1 Start Visual Basic.

2 In the New Project dialog box, select **Standard EXE** as the project type and click **Open**. This creates a new Visual Basic project with a blank form.

3 From the main menu, select **Project > References** to open the Project References dialog box.

4 Select **magicdemo 1.0 Type Library** from the list of available components and click **OK**.

5 Returning to the Visual Basic main menu, select **Project > Components** to open the Components dialog box.

6 Select **Microsoft Windows Common Controls 6.0** and click **OK**. You will use the `ListView` control from this component library.

## Creating the User Interface

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

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

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

```
Private Size As Double 'Holds current matrix size
Private theMagic As magicdemo.magicdemoclass 'magic object instance

Private Sub Form_Load()
'This function is called when the form is loaded.
'Creates a new magic class instance.
    On Error GoTo Handle_Error
    Set theMagic = New magicdemo.magicdemoclass
    Size = 0
Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub btnCreate_Click()
'This function is called when the Create button is pressed.
'Calls the mymagic method, and displays the magic square.
    Dim y As Variant
    If Size <= 0 Or theMagic Is Nothing Then Exit Sub
    On Error GoTo Handle_Error
    Call theMagic.mymagic(1, y, Size)
    Call ShowMatrix(y)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub edtSize_Change()
'This function is called when ever the contents of the
'Text box change. Sets the current value of Size.
    On Error Resume Next
    Size = CDbl(edtSize.Text)
    If Err <> 0 Then
        Size = 0
    End If
```

```
End Sub

Private Sub ShowMatrix(y As Variant)
'This function populates the ListView with the contents of
'y. y is assumed to contain a 2D array.
    Dim n As Long
    Dim i As Long
    Dim j As Long
    Dim nLen As Long
    Dim Item As ListItem

    On Error GoTo Handle_Error
    'Get array size
    If IsArray(y) Then
        n = UBound(y, 1)
    Else
        n = 1
    End If
    'Set up Column headers
    nLen = lstMagic.Width / 5
    Call lstMagic.ListItems.Clear
    Call lstMagic.ColumnHeaders.Clear
    Call lstMagic.ColumnHeaders.Add(, , "", nLen, lvwColumnLeft)
    For i = 1 To n
        Call lstMagic.ColumnHeaders.Add(, , _
        "Column " & Format(i), nLen, lvwColumnLeft)
    Next
    'Add array contents
    If IsArray(y) Then
        For i = 1 To n
            Set Item = lstMagic.ListItems.Add(, , "Row " & Format(i))
            For j = 1 To n
                Call Item.ListSubItems.Add(, , Format(y(i, j)))
            Next
        Next
    Else
        Set Item = lstMagic.ListItems.Add(, , "Row 1")
        Call Item.ListSubItems.Add(, , Format(y))
    End If
    Exit Sub
```

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

## Creating the Executable in Microsoft Visual Basic

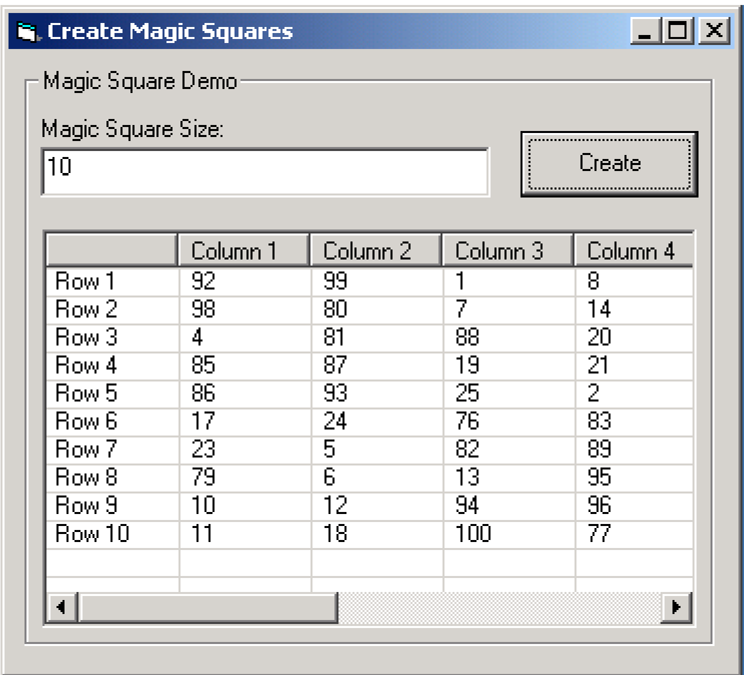
After the code is complete, create the standalone executable `magic.exe`:

1. Reopen the project by selecting **File > Save Project** from the main menu. Accept the default name for the main form and enter `magic.vbp` for the project name.

2. Return to the **File** menu. Select **File > Make magic.exe** to create the finished product.

## Testing the Application

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




| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| Row 1 | 92 | 99 | 1 | 8 |
| Row 2 | 98 | 80 | 7 | 14 |
| Row 3 | 4 | 81 | 88 | 20 |
| Row 4 | 85 | 87 | 19 | 21 |
| Row 5 | 86 | 93 | 25 | 2 |
| Row 6 | 17 | 24 | 76 | 83 |
| Row 7 | 23 | 5 | 82 | 89 |
| Row 8 | 79 | 6 | 13 | 95 |
| Row 9 | 10 | 12 | 94 | 96 |
| Row 10 | 11 | 18 | 100 | 77 |

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

## Packaging the Component

As a final step, package the `magicdemo` component and all supporting libraries into a self-extracting executable. Then anyone can install the package on another computer, in particular a computer without MATLAB installed, and use the `magicdemo` application.

To package the component:

1. Return to the Deployment Tool window and open the `magicdemo` project. If necessary, type `deploytool` in the Command Window.

2. Click the button in the toolbar.

The Deployment Tool creates the `magicdemo_pkg.exe` self-extracting executable.

To install the component onto another computer, copy the `magicdemo_pkg.exe` package to that machine, run `magicdemo_pkg.exe` from a command prompt, and follow the instructions.

# Creating an Excel Add-In: Spectral Analysis Example

**In this section...**


"Example Overview" on page 15-10

"Building the Component" on page 15-10

"Integrating the Component with VBA" on page 15-12

"Creating the Microsoft® Visual Basic Form" on page 15-14

"Adding the Spectral Analysis Menu Item to Microsoft® Excel" on page 15-20

"Saving the Add-In" on page 15-21

"Testing the Add-in" on page 15-21

"Packaging and Distributing the Add-In" on page 15-23


## Example Overview

This example shows how to create a comprehensive Microsoft Excel add-in to perform spectral analysis. It requires knowledge of Microsoft Visual Basic forms and controls, as well as Excel workbook events. See the Visual Basic documentation included with Excel 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 graphical user interface (GUI).

## Building the Component

Your component will have one class with the following two methods:

- The `computefft` method computes the FFT and power spectral density of the input data and computes a vector of frequency points based on the length of the data entered and the sampling interval.
- The `plotfft` method performs the same operations as `computefft`, but also plots the input data and the power spectral density in a MATLAB figure window.

The MATLAB code for these two methods resides in two MATLAB files, `computefft.m` and `plotfft.m`, as shown:

```
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 then zero');
        return;
    end
    fftdata = fft(data);
    freq = (0:length(fftdata)-1)/(length(fftdata)*interval);
    powerspect = abs(fftdata)/(sqrt(length(fftdata)));


plotfft.m:

function [fftdata, freq, powerspect] = plotfft(data, interval)
    [fftdata, freq, powerspect] = computefft(data, interval);
    len = length(fftdata);
    if (len <= 0)
        return;
    end
    t = 0:interval:(len-1)*interval;
    subplot(2,1,1), plot(t, data)
    xlabel('Time'), grid on
    title('Time domain signal')
    subplot(2,1,2), plot(freq(1:len/2), powerspect(1:len/2))
```

```
xlabel('Frequency (Hz)'), grid on
title('Power spectral density')
```

To build the component:

1. Start `deploytool`.
2. Create a new project with these settings:
   - **Project name**: `Fourier`
   - **Class name**: `Fourier`
3. Add the `computefft.m` and `plotfft.m` MATLAB files to the project.
4. Save the project.
5. Click the button in the toolbar to create the component.

## Integrating the Component with VBA

The next task is to implement the necessary VBA code to integrate the component into Excel.

To open Excel and select the libraries you need to develop the add-in:

1. Start Excel.
2. From the Excel main menu, select **Tools > Macro > Visual Basic Editor** to open the Visual Basic Editor.
3. Select **Tools > References** to open the Project References dialog box.
4. Select **Fourier 1.0 Type Library** and **MWComUtil 7.5 Type Library**.

### Creating the Main VBA Code Module

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

1. Right-click **VBAProject** 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
'
'Global instance of Fourier object
Public theFourier As Fourier.Fourier
'Global instance of MWComplex to accept FFT
Public theFFTData As MWComplex
'Input data range
Public InputData As Range
'Sampling interval
Public Interval As Double
'Output frequency data range
Public Frequency As Range
'Output power spectral density range
Public PowerSpect As Range
'Holds the state of plot flag
Public bPlot As Boolean
'Global instance of MWUtil object
Public theUtil as MWUtil
'Module-is-initialized flag
Public bInitialized As Boolean
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 theFourier Is Nothing Then
        Set theFourier = New Fourier.Fourier
    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 Microsoft Visual Basic Form

The next task is to develop a user interface for your add-in using the Microsoft Visual Basic editor. Follow these steps to create a new user form and populate it with the necessary controls:

1 Right-click **VBAProject** in the Project window and select **Insert > UserForm**.

   A new form appears under **Forms** in the **VBA Project**.

2 In the form's property page, set the **Name** property to `frmFourier` and the **Caption** property to `Spectral Analysis`.

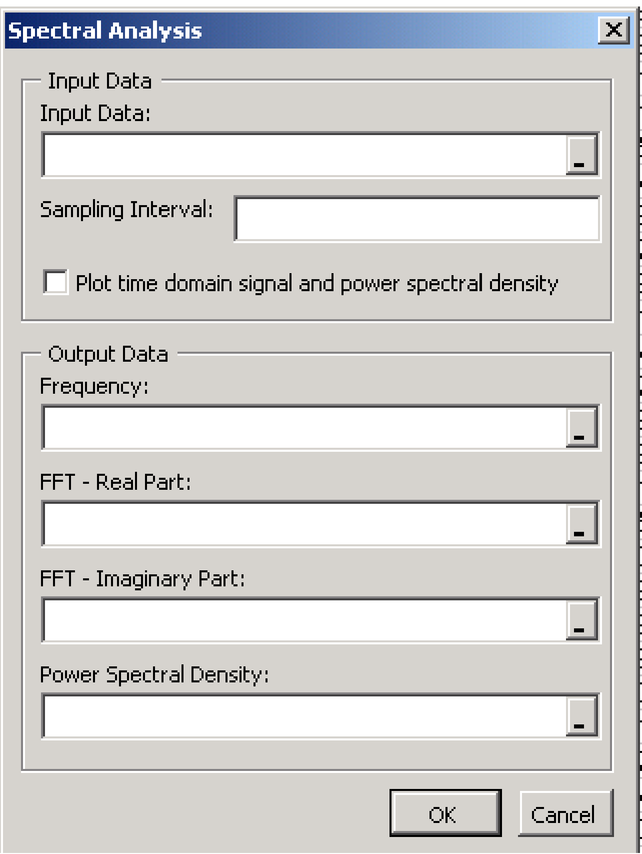
3 Add a series of controls to the blank form to complete the dialog box, as summarized in the following table:

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `Frame` | `Frame1` | Caption = `Input Data` | Groups all input controls. |
| `Label` | `Label1` | Caption = `Input Data:` | Labels `RefEdit` for input data. |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| RefEdit | refedtInput | | Selects range for input data. |
| Label | Label2 | Caption = Sampling Interval | Labels text box for sampling interval. |
| TextBox | edtSample | | Specifies the sampling interval. |
| CheckBox | chkPlot | Caption = Plot time domain Signal and Power Spectral Density | Plots input data and power spectral density. |
| Frame | Frame2 | Caption = Output Data | Groups all output controls. |
| Label | Label3 | Caption = Frequency: | Labels RefEdit for frequency output. |
| RefEdit | refedtFreq | | Selects output range for frequency points. |
| Label | Label4 | Caption = FFT - Real Part: | Labels RefEdit for real part of FFT. |
| RefEdit | refedtReal | | Selects output range for real part of FFT of input data. |
| Label | Label5 | Caption = FFT - Imaginary Part: | Labels RefEdit for imaginary part of FFT. |
| RefEdit | refedtImag | | Selects output range for imaginary part of FFT of input data. |
| Label | Label6 | Caption = Power Spectral Density | Labels RefEdit for power spectral density. |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| RefEdit | refedtPowSpect | | Selects the output range for power spectral density of input data. |
| CommandButton | btnOK | Caption = OK<br>Default = True | Executes the function and closes the dialog box. |
| CommandButton | btnCancel | Caption = Cancel<br>Cancel = True | Closes the dialog box without executing the function. |

The following figure shows the resulting layout.




**4** When the form and controls are complete, right-click anywhere in the form and **View Code**. The following code listing shows the code to implement. Note that this code references the control and variable names listed in the previous table. If you have renamed 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 theFFTData 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 (theFFTData.Real) Then
    If IsObject(theFFTData.Real) And TypeOf theFFTData.Real Is Range Then
            refedtReal.Text = theFFTData.Real.Address
        End If
    End If
    If Not IsEmpty (theFFTData.Imag) Then
    If IsObject(theFFTData.Imag) And TypeOf theFFTData.Imag Is Range Then
            refedtImag.Text = theFFTData.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 Microsoft Excel

The last task in the integration process is to add a menu item to Microsoft Excel so that you can invoke the tool from the Excel **Tools** menu. To do this you add event handlers for the workbook's `AddinInstall` and `AddinUninstall` events; these are 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 **ThisWorkbook** in the Visual Basic project window and select **View Code**.

2 Add the following code to the **ThisWorkbook** object:

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

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

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

    'Remove if already exists
    Call RemoveFourierMenuItem
    'Find Tools menu
    Set ToolsMenu = Application.CommandBars(1).FindControl(ID:=30007)
```

```
        If ToolsMenu Is Nothing Then Exit Sub
        'Add Spectral Analysis menu item
        Set NewMenuItem = ToolsMenu.Controls.Add(Type:=msoControlButton)
        NewMenuItem.Caption = "Spectral Analysis..."
        NewMenuItem.OnAction = "LoadFourier"
    End Sub

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

## Saving the Add-In

Name the add-in `Spectral Analysis` and follow these steps to save it:

1 From the Excel main menu, select **File > Properties**.

  The Workbook Properties dialog box opens.

2 Click the **Summary** tab and enter `Spectral Analysis` as the workbook title.

3 Click **OK** to save the edits.

4 Select **File > Save As** from the Excel main menu.

5 Select **Microsoft Excel Add-In (*.xla)** as the file type.

6 Enter `Fourier.xla` as the file name.

7 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.

To create the test problem:

1. Start a new Excel session with a blank workbook.
2. Select **Tools > Add-Ins** from the main menu.
3. When the Add-Ins dialog box opens, click **Browse**.
4. Browse to the `Fourier.xla` file 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 second. Put the time points into column A and the signal points into column B.

### Creating the Data

1. Enter `0` for cell A1 in the current worksheet.
2. Click cell A2 and type the formula `= A1 + 0.01`.
3. Drag the formula in cell A2 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 formula `SIN(2*PI()*15*A1) + SIN(2*PI()*40*A1) + RAND()`.

5. 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 **Input Data**.
3. Click the B1:B1001 range from the worksheet, or type this address into **Input Data**.
4. Click the **Sampling Interval** box and type `0.01`.
5. Click **Plot time domain signal and power spectral density**.
6. Enter `C1:C1001` for frequency output. Similarly, 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 following figure shows the output.

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

## Packaging and Distributing the Add-In

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

To package and distribute the add-in:

1. Return to the Deployment Tool and open the Fourier project. (If necessary run the `deploytool` command in the MATLAB product to reopen the Deployment Tool.)
2. Click the button in the toolbar.

The builder creates the `Fourier_pkg.exe` self-extracting executable.

3 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.

# Univariate Interpolation Example

**In this section...**

"Example Overview" on page 15-25

"Using the Deployment Tool to Create and Build the Component" on page 15-25

"Using the Component in Microsoft® Visual Basic" on page 15-26

"Creating the Microsoft® Visual Basic Form" on page 15-27

## Example Overview

This example is created using the Akima's Univariate Interpolation example posted by N. Shyamsundar on the MathWorks Web site. You can download the original MATLAB file from `http://www.mathworks.com/matlabcentral/`. Search for *COM Builder Example: Univariate Interpolation*.

This example shows you how to create the COM component using the MATLAB Builder NE product and how to use this COM component in external Microsoft Visual Basic code independent of the MATLAB product.

## Using the Deployment Tool to Create and Build the Component

1. At the MATLAB command prompt, change folders to your work folder.

2. Open the Deployment Tool window.

   ```
   deploytool
   ```

3. Create a project with the following settings:

| Setting | Value |
|---|---|
| Project name | `UnivariateInterp` |
| Class name | `Interp` |

| Setting | Value |
|---|---|
| Project folder | The name of your work folder followed by the Project name. |
| Generate Verbose Output | Selected |

4 Locate your work folder and navigate to the `UnivariateInterp` folder, and add the MATLAB file to the project.

5 Build the component by clicking the button in the Deployment Tool toolbar.

   The build process begins, and a log of the build appears in the **Output** pane of the Deployment Tool. The files that are needed for the component are copied to two newly created folders, `src` and `distrib`, in the `UnivariateInterp` folder. A copy of the build log is placed in the `src` folder.

## Using the Component in Microsoft Visual Basic

You can call the component from any application that supports COM.

To create a Microsoft Visual Basic project and add references to the necessary libraries:

1 Start Visual Basic.

2 Create a new Standard EXE project.

3 Select **Project > References**.

4 Ensure that the following libraries appear:

   `UnivariateInterp 1.0 Type Library`

   `MWComUtil 7.5 Type Library`

**Tip** If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to "Component Registration" on page 16-4 for information on this process.

## Creating the Microsoft Visual Basic Form

The next step creates a front end or a Microsoft Visual Basic form for the application. Your application receives data from the user through this form.

To create a new user form and populate it with the necessary controls.

1. Select **Projects > Component**. Alternatively, press **Ctrl+T**.
2. Ensure that **Microsoft Windows Common Controls 6.0** is selected.

   You will use the `ListView` control from this component library.
3. Add a series of controls to the blank form to create an interface using the properties shown in the following table.

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `Form` | `frmInterp` | Caption = `Univariate Interpolation` | Container for all components |
| `Label` | `lblDataCount` | Caption = `Number of Data Points` | Labels the text box `txtNumDataPts` |
| `TextBox` | `txtNumDataPts` | Text = | Number of original data points |
| `Label` | `lblInterp` | Caption = `Number of Interpolation Points` | Labels the text box `txtInterp` |
| `TextBox` | `txtInterp` | Text = | Number of points over which to interpolate |
| `Label` | `lblPlot` | Caption = `Would you like to plot the data?` | Labels the check box `chkPlot` |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `CheckBox` | `chkPlot` | | When selected, a message is sent to the COM component to plot the data. |
| `ListView` | `lstXData` | Name = `lstXData`<br>GridLines = `True`<br>LabelEdit = `lvwAutomatic`<br>View = `lvwReport` | X-data values. Set the view type to `lvwReport` to allow the user to add data to the list view. |
| `ListView` | `lstYData` | Name = `lstYData`<br>GridLines = `True`<br>LabelEdit = `lvwAutomatic`<br>View = `lvwReport` | Y-data values. Set the view type to `lvwReport` to allow the user to add data to the list view. |
| `ListView` | `lstInterp` | Name = `lstInterp`<br>GridLines = `True`<br>LabelEdit = `lvwAutomatic`<br>View = `lvwReport` | Interpolation points |
| `CommandButton` | `cmdEvaluate` | Caption = `Evaluate`<br>Default = `True` | Executes the function |
| `CommandButton` | `cmdCancel` | Caption = `Cancel`<br>Cancel = `True` | Closes the dialog box without executing function |

4 When the design is complete, save the project by selecting **File > Save**.

5 When prompted for the project name, type `Interp.vbp`, and for the form, type `frmInterp.frm`.

6 To write the underlying code, right-click **frmInterp** in the Project window and select **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Private theInterp As UnivariateInterp.Interp 'Variable to hold the COM object

Private Sub cmdCancel_Click()
    ' Unload the form if the user hits the cancel button.
    Unload Me
End Sub

Private Sub Form_Initialize()
    On Error GoTo Handle_Error
    ' Create the COM object
    ' If there is an error, handle it accordingly.
    Set theInterp = New UnivariateInterp.Interp
    ' Set the flags such that the input is always passed as double data.
    theInterp.MWFlags.DataConversionFlags.CoerceNumericToType = mwTypeDouble
    Exit Sub
Handle_Error:
    ' Error handling code
    MsgBox ("Error " & Err.Description)
End Sub

Private Sub Form_Load()
    ' Set the run time properties of the components
    Dim Len1 As Long ' Variable to hold length parameter of the list box
    Dim Len2 As Long ' Variable to hold the length parameter of the list box
    Len2 = lstInterp.Width / 2
    Len1 = (lstInterp.Width - Len2) - 150
    ' Add the column headers to the list boxes
    Call lstXData.ColumnHeaders.Add(, , "XData", Len2)
    Call lstYData.ColumnHeaders.Add(, , "YData", Len2)
    Call lstInterp.ColumnHeaders.Add(, , "Interp Data", Len1)
    Call lstInterp.ColumnHeaders.Add(, , "Interp YData", Len2)

    ' Enable the grid lines
    lstXData.GridLines = True
    lstYData.GridLines = True
```

```
    lstInterp.GridLines = True
    lstInterp.FullRowSelect = True

    ' Set the Tab indices for each of the components
    txtNumDataPts.TabIndex = 1
    txtInterp.TabIndex = 2
    lstXData.TabIndex = 3
    lstYData.TabIndex = 4
    lstInterp.TabIndex = 5
    cmdEvaluate.TabIndex = 6
    cmdCancel.TabIndex = 7
End Sub

Private Sub txtInterp_Change()
    ' If user changes number of interpolation points, set the interpolation
    ' point listbox to accomodate the new number of points.
    Dim loopCount As Integer ' loop count
    Dim numData As Integer
    On Error GoTo Handle_Error
    ' First clear the listbox
    Call lstInterp.ListItems.Clear
    ' Create space for the requested number of interpolation points
    If Not (txtInterp.Text = "") Then
        numData = CDbl(txtInterp.Text)
        For loopCount = 1 To numData
            Call lstInterp.ListItems.Add(loopCount, , "")
        Next
    End If
    Exit Sub
Handle_Error:
    ' Reset the list to 0 elements and also the text box to an empty string.
    MsgBox ("Invalid value for number of Data points")
    lstInterp.ListItems.Clear
    txtInterp.Text = ""
End Sub

Private Sub txtNumDataPts_Change()
    ' If the user changes the number of data points, set the XData and YData
    ' listboxes to accomodate the new number of points.
    Dim loopCount As Integer ' loop count
```

```
    Dim numData As Integer
    On Error GoTo Handle_Error
    ' First clear both the listbox (XData and YData)
    Call lstXData.ListItems.Clear
    Call lstYData.ListItems.Clear
    ' Create space for the requested number of data points (XData and YData).
    If Not (txtNumDataPts.Text = "") Then
        numData = CDbl(txtNumDataPts.Text)
        For loopCount = 1 To numData
            Call lstXData.ListItems.Add(loopCount, , "")
            Call lstYData.ListItems.Add(loopCount, , "")
        Next
    End If
    Exit Sub
Handle_Error:
    ' Reset the list to 0 elements and also the text box to an empty string.
    MsgBox ("Error: " & Err.des)
    Call lstXData.ListItems.Clear
    Call lstYData.ListItems.Clear
    txtNumDataPts.Text = ""
End Sub

Private Sub cmdEvaluate_Click()
    ' Dim R As Range
    Dim XDataInterp As Variant ' Result variable object
    Dim loopCount As Integer ' A variable used for loop count
    Dim item As ListItem ' Temporary variable to store data in list box
    Dim XData() As Double ' X value of data points, passed to COM object
    Dim YData() As Double ' Y value of data points, passed to the COM object
    Dim XInterp() As Double ' X value of interpolation points, passed to COM
                            ' object
    Dim Yi As Variant ' Y value of interpolation points, obtained from COM
                      ' object as ouput value

    ' Set dimensions of the input and ouput data based on user inputs (number
    ' of data points and number of interpolation points).
    ReDim XData(1 To lstXData.ListItems.Count)
    ReDim YData(1 To lstYData.ListItems.Count)
    ReDim XInterp(1 To lstInterp.ListItems.Count)
    ReDim Yi(1 To lstInterp.ListItems.Count)
```

```
    ' Collect the Data and set the XData, YData, XInterp matrices accordingly
    For loopCount = 1 To lstXData.ListItems.Count
        XData(loopCount) = CDbl(lstXData.ListItems.item(loopCount))
        YData(loopCount) = CDbl(lstYData.ListItems.item(loopCount))
    Next
    For loopCount = 1 To lstInterp.ListItems.Count
        XInterp(loopCount) = CDbl(lstInterp.ListItems.item(loopCount))
        Yi(loopCount) = -1
    Next

    ' Check if the object was created properly.
    ' If not, go to the error handling routine.

    If theInterp Is Nothing Then GoTo Exit_Form

    ' If there is an error, continue with the code.
    On Error GoTo Handle_Error

    'Compute Curve Fitting Data
    Call theInterp.UnivariateInterpolation(1,Yi,XData,YData,XInterp,_
                                           chkPlot.Value)

    'Call lstInterp.ListItems.Clear
    For loopCount = LBound(Yi, 2) To UBound(Yi, 2)
        Set item = lstInterp.ListItems(loopCount)
        Call item.ListSubItems.Add(, , Format(Yi(1, loopCount), "##.###"))
    Next
    Call lstInterp.Refresh
    GoTo Exit_Form
Handle_Error:
    ' Error handling routine
    MsgBox ("Error: " & Err.Description)
Exit_Form:
End Sub
```

# Matrix Calculator Example

**In this section...**

"Example Overview" on page 15-33

"Building the Component" on page 15-33

"Using the Component in Microsoft® Visual Basic" on page 15-34

"Creating the Microsoft® Visual Basic Form" on page 15-35

## Example Overview

This example shows how to encapsulate MATLAB utilities that perform basic matrix arithmetic. It includes MATLAB code that performs matrix addition, subtraction, multiplication, division and left division and a function to evaluate the eigenvalues for a matrix. The example shows how to create the COM component using the MATLAB Builder NE product and how to use the COM component in a Microsoft Visual Basic application independent of the MATLAB product.

---

**Note** This example assumes that you have downloaded the MATLAB code from `http://www.mathworks.com/matlabcentral/` to your work folder. To get the download, search the `File Exchange` at `matlabcentral` for `MatrixArith`.

---

## Building the Component

1. At the MATLAB command prompt, change folders to the MatrixMath folder in your work folder.

2. Enter the command `deploytool` to open the Deployment Tool window.

3. Create a project with the following settings:

| Setting | Value |
|---|---|
| Project name | `matrixMath` |
| Class name | `matrixMathclass` |
| Project folder | The name of your work folder followed by the project name |
| Generate Verbose Output | Selected |

4 Locate your work folder and navigate to the `matrixMath` folder, which contains the MATLAB files needed for the component.

5 Add the following files to the project:

- `addMatrices.m`
- `divideMatrices.m`
- `eigenValue.m`
- `leftDivideMatrices.m`
- `multiplyMatrices.m`
- `subtractMatrices.m`

6 Build the component by clicking the button in the Deployment Tool toolbar.

The build process begins, and a log of the build appears in the **Output** pane of the Deployment Tool. The files that are needed for the component are copied to two newly created folders, `src` and `distrib`, in the `matrixMath` folder. A copy of the build log is placed in the `src` folder.

## Using the Component in Microsoft Visual Basic

You can call the component from any application that supports COM. Follow these steps to create a Microsoft Visual Basic project and add references to the necessary libraries.

1 Start Visual Basic.

2 Create a new Standard EXE project.

3 Select **Project > References**.

4 Ensure that the following libraries are in the project:

```
MatrixMath 1.0 Type Library

MWComUtil 7.5 Type Library
```

---

**Note** If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to "Component Registration" on page 16-4 for information on this.

---

## Creating the Microsoft Visual Basic Form

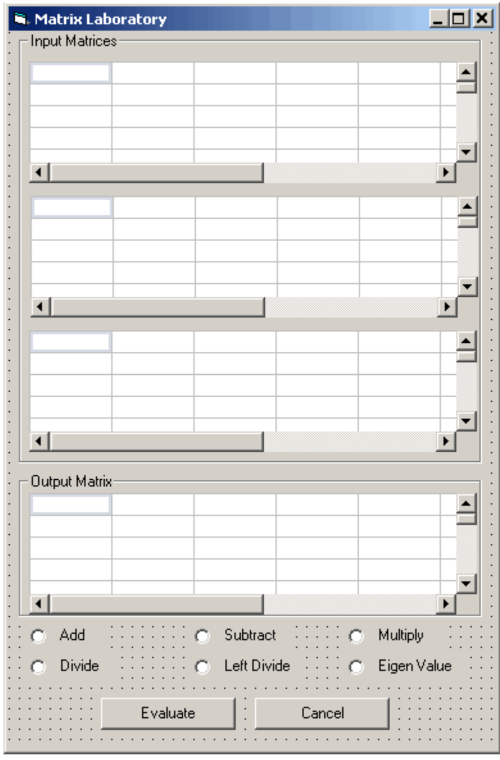
The next step creates a front end or a Microsoft Visual Basic form for the application. End users enter data in this form.

To create a new user form and populate it with the necessary controls:

1 Select **Projects > Component**. Alternatively, press **Ctrl+T**.

2 Make sure that **Microsoft Windows Common Controls 6.0** is selected. You will use the `Spreadsheet` control from this component library.

3 Add a series of controls to the blank form to create an interface as shown in the next figure.




4 One of the main components used in the Visual Basic form is a `Spreadsheet` component. For each `Spreadsheet` component, set properties as follows.

| Property | Original Value | New Value |
|---|---|---|
| DisplayColumnHeaders | True | False |
| DisplayHorizontalScrollBar | True | False |
| DisplayRowHeaders | True | False |
| DisplayTitleBar | True | False |
| DisplayToolBar | True | False |
| DisplayVerticalScrollBar | True | False |
| MaximumWidth | 80% | 100% |
| ViewableRange | 1:65536 | A1:E5 |

A consolidated list of components added to the form and the properties modified is as follows.

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| Form | frmMatrixMath | Caption = Matrix Laboratory | Container for all components |
| Frame | frmInput | Caption = Input Data Points | Groups all input controls |
| Frame | frmOutput | Caption = Output Coefficients | Groups all output controls |
| Spreadsheet | sheetMat1 | Refer to previous table. | Accepts input matrix 1 from user |
| Spreadsheet | sheetMat2 | Refer to previous table. | Accepts input matrix 2 from user |
| Spreadsheet | sheetMat3 | Refer to previous table. | Accepts input matrix 3 from user |
| Spreadsheet | sheetResultMat | Refer to previous table. | Displays result matrix |
| Label | lblAdd | Caption = Add | Labels **Add** option button |
| OptionButton | optOperation | Index = 0 | Option button to perform addition |
| Label | lblSub | Caption = Subtract | Labels **Subtract** option button |
| OptionButton | optOperation | Index = 1 | Option button to perform subtraction |
| Label | lblMult | Caption = Multiply | Labels **Multiply** option button |
| OptionButton | optOperation | Index = 2 | Option button to perform multiplication |
| Label | lblDivide | Caption = Divide | Labels **Divide** option button |
| OptionButton | optOperation | Index = 3 | Option button to perform division |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `Label` | `lblLeftDivide` | Caption = `Left Divide` | Labels **Left Divide** option button |
| `OptionButton` | `optOperation` | Index = `4` | Option button to perform left division |
| `Label` | `lblEig` | Caption = `Eigenvalue` | Labels **Eigenvalue** option button |
| `OptionButton` | `optOperation` | Index = `5` | Option button to calculate Eigenvalue of first matrix |
| `CommandButton` | `cmdEvaluate` | Caption = `Evaluate`<br>Default = `True` | Executes function |
| `CommandButton` | `cmdCancel` | Caption = `Cancel`<br>Cancel = `True` | Closes dialog box without executing function |

**5** When the design is complete, save the project by selecting **File > Save**. When prompted for the project name, type `MatrixMathVB.vbp`, and for the form, type `frmMatrixMath.frm`.

**6** To write the underlying code, right-click **frmMatrixMath** in the Project window, and select **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Dim theMatCal As matrixMath.matrixMath

Private Sub Form_Initialize()
' Create an instance of the COM object and set the MWArray flags.
' If this fails, exit from the form.
On Error GoTo exit_form
    ' Create the object.
    Set theMatCal = New matrixMath.matrixMath
```

```
    ' Force the input to be of type double.
    theMatCal.MWFlags.DataConversionFlags.CoerceNumericToType = mwTypeDouble
    ' Set the AutoResizeOutput flag to True, so that you do not have to specify
    ' the size of the output variable as returned by the COM object.
    theMatCal.MWFlags.ArrayFormatFlags.AutoResizeOutput = True
    ' Get the results in a Matrix format.
    theMatCal.MWFlags.ArrayFormatFlags.OutputArrayFormat =_
mwArrayFormatMatrix
    Exit Sub
exit_form:
    ' Error handling routine. Since no object is created, display error '
    'message and unload the form.
    MsgBox ("Error: " & Err.Description)
    Unload Me
End Sub

Private Sub Form_Load()
    ' Set the run time properties for all the components.
    frmInputs.TabIndex = 1
    sheetMat1.AutoFit = True

    ' Set the tab order for each component and the viewable range.
    ' If you need a larger viewable range, you might want to turn the
    ' horizontal and vertical scroll bars to TRUE.
    sheetMat1.TabStop = True
    sheetMat1.TabIndex = 1
    sheetMat1.Width = 4875
    sheetMat1.ViewableRange = "A1:E5"

    sheetMat2.TabStop = True
    sheetMat2.TabIndex = 2
    sheetMat2.Width = 4875
    sheetMat2.ViewableRange = "A1:E5"

    sheetMat3.TabStop = True
    sheetMat3.TabIndex = 3
    sheetMat3.Width = 4875
    sheetMat3.ViewableRange = "A1:E5"

    sheetResultMatTabStop = False
```

```
    sheetResultMatTabIndex = 1
    sheetResultMatWidth = 4875
    sheetResultMat.ViewableRange = "A1:E5"

    frmOutput.TabIndex = 2
    optOperation(0).TabIndex = 3
    optOperation(1).TabIndex = 4
    optOperation(2).TabIndex = 5
    optOperation(3).TabIndex = 6
    optOperation(4).TabIndex = 7
    optOperation(5).TabIndex = 8
End Sub

Private Sub cmdCancel_Click()
    ' When the user clicks on the Cancel button, unload the form.
    Unload Me
End Sub

Private Sub cmdEval_Click()
    ' Declare the variables to be used in the code
    Dim data1 As Range
    ' This is the temporary variable that holds the value entered in
    ' the spreadsheet.

    'Dim finalRows As Double ' The number of
    'Dim finalCols As Double

    ' Dim tempVal As Double
    Dim matArray1 As Variant ' Variable to hold the value of input Matrix 1,
                             ' passed to the COM object directly.
    Dim matArray2 As Variant ' Variable to hold the value of input Matrix 1,
                             ' passed via varArg variable.
    Dim matArray3 As Variant ' Variable to hold the value of input Matrix 1,
                             ' passed via varArg variable.
    Dim varArg(2) As Variant ' Variable to hold the value of input Matrix 1,,
    ' contains the two optional matrices and is passed to the COM object.

    'Dim mat1() As Double
    'Dim mat1Dimension2() As Variant
```

```
Dim tempRange As Range ' Take the range value as obtained from the
                       ' user input into a temporary range.
Dim resultMat As Variant ' Variable to take the result matrix in
Dim msg As String ' The message thrown by the COM object is taken
                  ' in this variable.

Call sheetResultMat.ActiveSheet.UsedRange.Clear

' Check if the COM object was created properly.
' If not exit
If theMatCal Is Nothing Then GoTo exit_form

' Get the used range of data from the sheetMat1, which will then be
' converted into matArray1.
Set data1 = sheetMat1.ActiveSheet.UsedRange

'finalRows = data1.Rows.Count
'finalCols = data1.Columns.Count

'ReDim mat1(1 To data1.Rows.Count)
'ReDim mat1Dimension2(1 To data1.Columns.Count)
ReDim matArray1(1 To data1.Rows.Count, 1 To data1.Columns.Count) As_
Double
For RowCount = 1 To data1.Rows.Count
    For ColCount = 1 To data1.Columns.Count
        ' Extract the values and populate input matrix 1.
        Set tempRange = data1.Cells(RowCount, ColCount)
        'tempVal = tempRange.Value
        'matArray1(RowCount, ColCount) = tempVal
        matArray1(RowCount, ColCount) = tempRange.Value
        'Set mat1(ColCount) = tempRange.Value
    Next ColCount
    ' mat1Dimension2(RowCount) = mat1()
Next RowCount

Set data1 = sheetMat2.ActiveSheet.UsedRange
If (Not (data1.Value = "")) Then
    ReDim matArray2(1 To data1.Rows.Count, 1 To data1.Columns.Count) As_
    Double
    For RowCount = 1 To data1.Rows.Count
```

```
            For ColCount = 1 To data1.Columns.Count
                Set tempRange = data1.Cells(RowCount, ColCount)
                tempVal = tempRange.Value
                matArray2(RowCount, ColCount) = tempVal
            Next ColCount
        Next RowCount
        finalCols = data1.Columns.Count
        varArg(0) = matArray2
    End If

    Set data1 = sheetMat3.ActiveSheet.UsedRange
    If (Not (data1.Value = "")) Then
        ReDim matArray3(1 To data1.Rows.Count, 1 To data1.Columns.Count) As_
Double
        For RowCount = 1 To data1.Rows.Count
            For ColCount = 1 To data1.Columns.Count
                Set tempRange = data1.Cells(RowCount, ColCount)
                tempVal = tempRange.Value
                matArray3(RowCount, ColCount) = tempVal
            Next ColCount
        Next RowCount
        finalCols = data1.Columns.Count
        varArg(1) = matArray3
    End If

    ' Based on the operation selected by the user, call the appropriate method
    ' from the COM object.
    If optOperation.Item(0).Value = True Then ' Add
        Call theMatCal.addMatrices(2, resultMat, msg, matArray1, varArg)
    ElseIf optOperation.Item(1).Value = True Then ' Subtract
        Call theMatCal.subtractMatrices(2, resultMat, msg, matArray1, varArg)
    ElseIf optOperation.Item(2).Value = True Then ' Multiply
        Call theMatCal.multiplyMatrices(2, resultMat, msg, matArray1, varArg)
    ElseIf optOperation.Item(3).Value = True Then ' Divide
        Call theMatCal.divideMatrices(2, resultMat, msg, matArray1, varArg)
    ElseIf optOperation.Item(4).Value = True Then ' Left Divide
        Call theMatCal.leftDivideMatrices(2, resultMat, msg, matArray1,_
        varArg)
    ElseIf optOperation.Item(5).Value = True Then ' Eigen Value
        Call theMatCal.eigenValue(2, resultMat, msg, matArray1)
```

```
    End If

    ' If the result matrix is a scalar double, display it in the first cell.
    If (VarType(resultMat) = vbDouble) Then
        Set tempRange = sheetResultMat.Cells(1, 1)
        tempRange.Value = resultMat

    ' If the result matrix is not a scalar double, loop through it to display
    ' all the elements.
    Else
        For RowCount = 1 To UBound(resultMat, 1)
            For ColCount = 1 To UBound(resultMat, 2)
                Set tempRange = sheetResultMat.Cells(RowCount, ColCount)
                tempRange.Value = resultMat(RowCount, ColCount)
            Next ColCount
        Next RowCount
    End If
    Exit Sub
exit_form:
    MsgBox ("Error: " & Err.Description)
    Unload Me
End Sub

' If the user changes the operation, clear the result matrix.
Private Sub optOperation_Click(Index As Integer)
    Call sheetResultMat.ActiveSheet.Cells.Clear
End Sub
```

# Curve Fitting Example

**In this section...**


"Example Overview" on page 15-44

"Building the Component" on page 15-44

"Building the Project" on page 15-45

"Using the Component in Microsoft® Visual Basic" on page 15-45

"Creating the Microsoft® Visual Basic Form" on page 15-45


## Example Overview

This example demonstrates the optimal fitting of a nonlinear function to a set of data, using the curve-fitting demo `fitfun` provided with the MATLAB product. It uses `fminsearch`, an implementation of the Nelder-Mead simplex (direct search) algorithm, to minimize a nonlinear function of several variables.

This example shows you how to create the COM component using the MATLAB Builder NE product and how to use this COM component in a Microsoft Visual Basic application independent of MATLAB.

---

**Note** This example assumes that you have downloaded the MATLAB code from `http://www.mathworks.com/matlabcentral/` to the *`matlabroot`* folder. To get the download, search the `File Exchange` at `matlabcentral` for `COM Builder Demo: Curve Fitting`.

---

## Building the Component

1. At the MATLAB command prompt, change folders to *`matlabroot`*.

2. Enter the `deploytool` command to open the Deployment Tool window.

3. Create a project with the following settings:

| Project name | CurveFit |
|---|---|
| Class name | CurveFitclass |

## Building the Project

1. In the Deployment Tool window, add `fitfun.m` and `fitdemo.m` from the folder *`matlabroot`*`/CurveFitDemo`.

2. Click the button in the toolbar.

   The component is created and placed in the `distrib` folder within the `Class`folder.

## Using the Component in Microsoft Visual Basic

You can call the component from any application that supports COM.

To create a Microsoft Visual Basic project and add references to the necessary libraries:

1. Start Visual Basic.

2. Create a new Standard EXE project.

3. Select **Project > References**.

4. Ensure that the following libraries are included in the project:

```
CurveFit 1.0 Type Library
MWComUtil 7.5 Type Library
```

---

**Note** If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to “Component Registration” on page 16-4 for information.

---

## Creating the Microsoft Visual Basic Form

The next step is to create a front end or a Microsoft Visual Basic form for the application. End users enter data on the form.

To create a new user form and populate it with the necessary contros:

1 Select **Projects > Component**. Alternatively, press **Ctrl+T**.

2 Make sure that **Microsoft Windows Common Controls 6.0** is selected. You will use the `ListView` control from this component library.

3 Add a series of controls to the blank form to create an interface.

   The following table shows the components and properties that are required.

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `Form` | `frmCurveFit` | Caption = `Curve Fitting` | Container for all components. |
| `Frame` | `frmInput` | Name = `frmInput*`<br>Caption = `Input Data Points` | Groups all input controls. |
| `Frame` | `frmOutput` | Name = `frmOutput*`<br>Caption = `Output Coefficients` | Groups all output controls. |
| `Label` | `lblNumDataPoints` | Caption = `Number of Data Points` | Labels the text box that takes the number of data points the user wants to enter. |
| `TextBox` | `txtNumOfDatPoints` | Text = | Holds number of data points the user wants to enter. Sets size of list box added later. |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| ListView | lstXData | Name = lstXData<br>GridLines = TrueLabel<br>Edit = lvwAutomatic<br>View = lvwReport | X-data values. Set the view type to lvwReport to enable user to add data to the list view. |
| ListView | lxtYData | Name = lstYData<br>GridLines = TrueLabel<br>Edit = lvwAutomatic<br>View = lvwReport | Y-data values. |
| Label | lblCoeff1* | Caption = Co-efficient 1 | Labels text box for coefficient 1. |
| Label | lblCoeff2 | Caption = Co-efficient 2 | Labels text box for coefficient 2. |
| TextBox | txtCoeff1 | Text = | Displays value of coefficient 1 as calculated by the COM module. |
| TextBox | txtCoeff2 | Text = | Displays value of coefficient 2 as calculated by the COM module. |
| Label | lblLambda1* | Caption = Lambda 1 | Labels text box for lambda 1. |
| Label | lblLambda2 | Caption = Lambda 2 | Labels text box for lambda 2. |
| TextBox | txtLambda1 | Text = | Displays value of lambda 1 as calculated by the COM module. |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `TextBox` | `txtLambda2` | Text = | Displays value of lambda 2 as calculated by the COM module. |
| `CommandButton` | `cmdEvaluate` | Caption = `Evaluate`<br>Default = `True` | Executes function. |
| `CommandButton` | `cmdCancel` | Caption = `Cancel`<br>Cancel = `True` | Closes dialog box without executing the function. |

4 When the design is complete, save the project by selecting **File > Save**.

5 When prompted for the project name, type `CurveFitExample.vbp`, and for the form, type `frmCurveFit.frm`.

6 In the Project window, right-click `frmCurveFit` and select **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Dim theFit As CurveFit.CurveFit ' Variable to hold the COM Object

' This routine is exectued when the form is initialized.
Private Sub Form_Initialize()
' If the initialize routine fails, handle it accordingly.
On Error GoTo Exit_Form
    ' Create the COM object
    Set theFit = New CurveFit.CurveFit
    ' Set the flags such that the output is transposed.
    theFit.MWFlags.ArrayFormatFlags.TransposeOutput = True
    Exit Sub
Exit_Form:
    ' Display the error message and Unload the form if object
```

```
creation failed
    MsgBox ("Error: " & Err.Description)
    MsgBox ("Error: Could not create the COM object")
    Unload Me
End Sub

Private Sub Form_Load()
On Error GoTo Exit_Form
    ' Set the runtime properties of the components

    ' Set the headers of the column
    Call lstXData.ColumnHeaders.Add(, , "X Data")
    Call lstYData.ColumnHeaders.Add(, , "Y Data")

    ' Make labeledit property automatic so that you edit the label.
    lstXData.LabelEdit = lvwAutomatic
    lstYData.LabelEdit = lvwAutomatic

    ' Make the grid lines for the listbox visible.
    lstXData.GridLines = True
    lstYData.GridLines = True
    Exit Sub
Exit_Form:
    ' Error handling routine. Since cannot load the form,
    ' display the error message and unload the program.
    MsgBox ("Error: Could not load the form")
    MsgBox ("Error: " & Err.Description)
    Unload Me
End Sub

Private Sub cmdCancel_Click()
    ' If the user hits the cancel button, unload the form.
    Unload Me
End Sub

Private Sub txtNumOfDataPoints_Change()
    ' If user changes number of data points, clear XData and YData
    ' listboxes. Provide enough spaces for given number of points.
    Dim loopCount As Integer
    Call lstXData.ListItems.Clear
```

```
    Call lstYData.ListItems.Clear
    If (txtNumOfDataPoints.Text = "") Then
        Exit Sub
    End If
    For loopCount = 1 To CInt(txtNumOfDataPoints.Text)
        lstXData.ListItems.Add (loopCount)
        lstYData.ListItems.Add (loopCount)
    Next loopCount
End Sub

Private Sub cmdEvaluate_Click()
    Dim loopCount As Integer ' loop counter
    Dim numOfData As Integer ' variable to hold the number of data
                             ' points the user has entered
    Dim XData() As Double ' Column Vector for XData, will be passed
                          ' as input to the COM method.
    Dim YData() As Double ' Column Vector for YData, will be passed
                          ' as input to the COM method.
    Dim Coeff As Variant ' Coefficient values will be returned by
                         ' the COM method in this variable.
    Dim Lambda As Variant ' Lambda values will be returned by the
                          ' COM method in this variable.

    ' If there is an error, handle it accordingly.
On Error GoTo Handle_Error
    If txtNumOfDataPoints.Text = "" Then
        Exit Sub
    End If
    ' Get the number of data points.
    numOfData = CInt(txtNumOfDataPoints.Text)
    ReDim XData(1 To numOfData) As Double
    ReDim YData(1 To numOfData) As Double
    ' Read the input data into respective double arrays.
    For loopCount = 1 To numOfData
        XData(loopCount) = lstXData.ListItems.Item(loopCount)
        YData(loopCount) = lstYData.ListItems.Item(loopCount)
    Next loopCount

    ' Call the COM method
    Call theFit.fitdemo(2, Coeff, Lambda, XData, YData)
```

```
    ' Display values of coefficients returned by the COM method.
    txtCoeff1.Text = CStr(Format(Coeff(1, 1), "##.####"))
    txtCoeff2.Text = CStr(Format(Coeff(1, 2), "##.####"))
    txtLambda1.Text = CStr(Format(Lambda(1, 1), "##.####"))
    txtLambda2.Text = CStr(Format(Lambda(1, 2), "##.####"))
    Exit Sub
Handle_Error:
    ' Error handling routine
    MsgBox ("Error: " & Err.Description)
End Sub
```

# Bouncing Ball Simulation Example

**In this section...**

"Example Overview" on page 15-52
"Building the Component" on page 15-52
"Using the Component in Microsoft® Visual Basic" on page 15-53
"Creating the Microsoft® Visual Basic Form" on page 15-54

## Example Overview

This example is adapted from the `ballode` demo provided with the MATLAB product. It demonstrates repeated event location, where the conditions are changed after each terminal event.

This demo computes 10 bounces with calls to `ode23`, which is a MATLAB function. A user-specified damping factor after each bounce attenuates the speed of the ball. The trajectory is plotted using the output function `odeplot`. In addition to the damping factor, the user can also provide the initial velocity, the maximum number of bounce to track, and the maximum time until demo is completed.

This example shows you how to create the COM component using the MATLAB Builder NE product and how to use this COM component in a Visual Basic application independent of MATLAB.

---

**Note** This example assumes that you have downloaded the MATLAB code to the *`matlabroot`* folder.

---

## Building the Component

1. At the MATLAB command prompt, change folders to *`matlabroot`*`/BallODE`.
2. Enter the command `deploytool` to open the Deployment Tool window.
3. Use the Deployment Tool to create a project with the following settings:

| Setting | Value |
|---|---|
| Project name | `bouncingBall` |
| Class name | `bouncingBallclass` |
| Project folder | The name of your work folder followed by the component name |
| Generate Verbose Output | Selected |

**4** Locate your work folder, navigate to `matlabroot/BallODE`, and add `ballode.m` to the project.

**5** Build the component by clicking the button in the Deployment Tool toolbar.

The build process begins, and a log of the build appears in the **Output** pane of the Deployment Tool window. The files that are needed for the component are copied to two newly created folders, `src` and `distrib`, in the `bouncingBall` folder. A copy of the build log is placed in the `src` folder.

## Using the Component in Microsoft Visual Basic

You can call the component from any application that supports COM.

To create a Microsoft Visual Basic project and add references to the necessary libraries:

**1** Start Visual Basic.

**2** Create a new Standard EXE project.

**3** Select **Project > References**.

**4** Select the following libraries:

- `bouncingBall 1.0 Type Library`

  (If you named your class something other than `bouncingBall` or gave a different version number, click and use the appropriate component and corresponding type library.)

- `MWComUtil 7.5 Type Library`

---

**Note** If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to "Component Registration" on page 16-4 for information on this.

---

## Creating the Microsoft Visual Basic Form

The next task is to create a front end or a Microsoft Visual Basic form for the application. End users enter data with this form.

To create a new user form and populate it with the necessary controls:

1. Select **Projects > Component**. Alternatively, press **Ctrl+T**.
2. Check that **Microsoft Windows Common Controls 6.0** is selected. You will use the `ListView` control from this component library.
3. Add a series of controls to the blank form to create an interface with the properties listed in the following table.

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `Form` | `frmBallOde` | Caption = `Bouncing Ball ODE` | Container for all components. |
| `Frame` | `frmInput` | Name = `frmInput*`<br>Caption = `Input Data Points` | Groups all input controls. |
| `Frame` | `frmOutput` | Name = `frmOutput*`<br>Caption = `Output Coefficients` | Groups all output controls. |
| `Label` | `lblInitVal` | Caption = `Initial Velocity` | Labels the text box `txtInitVal`. |
| `TextBox` | `txtInitVal` | Text = | Holds initial velocity by which ball is thrown into the air. |

| Control Type | Control Name | Properties | Purpose |
|---|---|---|---|
| `Label` | `lblDamp` | Caption = `Damping Factor` | Labels the text box `txtDamp`. |
| `TextBox` | `txtDamp` | Text = | Holds damping factor for the bounce, that is, the factor by which the speed of the ball is reduced after it bounces. |
| `Label` | `lblIter` | Caption = `Number of Bounces` | Labels the text box `txtIter`. |
| `TextBox` | `txtIter` | Text = | Holds number of iterations or bounces to track. |
| `Label` | `lblFinalTime` | Caption = `Maximum Time` | Labels the text box `txtFinalTime`. |
| `TextBox` | `txtFinalTime` | Text = | Stores time until demo is completed. |
| `ListView` | `lstBounce` | Name = `lstBounce`<br>GridLines = `True`<br>LabelEdit = `lvwManual`<br>View = `lvwReport` | Displays the time stamp when ball bounces off the ground. |
| `CommandButton` | `cmdEvaluate` | Caption = `Evaluate`<br>Default = `True` | Executes the function. |
| `CommandButton` | `cmdCancel` | Caption = `Cancel`<br>Cancel = `True` | Closes the dialog box without executing the function. |

4. When the design is complete, save the project by selecting **File > Save**. When prompted for the project name, type `BallOde.vbp`, and for the form, type `frmBallOde.frm`.

5. In the Project dialog box, right-click `frmBallOde` and select **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Private theBall As Variant ' Variable to hold the COM object.

Private Sub cmdCancel_Click()
    ' If the user presses the Cancel button, unload the form.
    Unload Me
End Sub

Private Sub Form_Initialize()
    Dim Len1 As Long ' Used to set length of columns for list box.
    Dim Len2 As Long ' Used to set length of columns for list box.
    On Error GoTo Handle_Error
    ' Set length of the each column based on length of the listbox
    ' such that the two columns span the maximum area without
    ' creating a horizontal scroll bar.
    Len2 = lstBounce.Width / 2
    Len1 = (lstBounce.Width - Len2) - 300

    ' Add column headers to each column in the list box.
    Call lstBounce.ColumnHeaders.Add(, , "Bounce", Len1)
    Call lstBounce.ColumnHeaders.Add(, , "Time", Len2)

    ' Set tab indices for each component.
    txtInitVel.TabIndex = 1
    txtDamp.TabIndex = 2
    txtIter.TabIndex = 3
    txtFinalTime.TabIndex = 4
    cmdEvaluate.TabIndex = 5
    cmdCancel.TabIndex = 6
    lstBounce.TabStop = False

    ' Create the COM object
    ' If there is an error, handle it accordingly.
    Set theBall = CreateObject("bouncingBall.bouncingBall.1_0")
    Exit Sub
Handle_Error:
```

```
    ' Error handling code
    MsgBox ("Error " & Err.Description)
End Sub
Private Sub cmdEvaluate_Click()
    ' Dim R As Range
    Dim zeroTime As Variant ' Result variable object.
    Dim loopCount As Integer
    Dim item As ListItem

    ' Check if the object was created properly.
    ' If not, go to the error handling routine.
    If theBall Is Nothing Then GoTo Exit_Form

    ' If there is an error, continue with the code.
    On Error Resume Next

    ' Process inputs
    ' If the user does not provide the values for input parameters,
    ' use the default values.
    If txtDamp.Text = Empty Then
        txtDamp.Text = 0.9
    End If
    If txtInitVel.Text = Empty Then
        txtInitVel.Text = 20
    End If
    If txtIter.Text = Empty Then
        txtIter.Text = 15
    End If
    If txtFinalTime.Text = Empty Then
        txtFinalTime.Text = 20
    End If

    'Compute Bouncing ball data
    Call theBall.ballode(1, zeroTime, CDbl(txtIter.Text),_
    CDbl(txtDamp.Text), CDbl(txtFinalTime.Text),_
    CDbl(txtInitVel.Text))

    ' Display the output values (time stamp when ball bounces on
    ' the ground).
    Call lstBounce.ListItems.Clear
```

```
    For loopCount = LBound(zeroTime, 1) To UBound(zeroTime, 1)
        Set item = lstBounce.ListItems.Add(, , Format(loopCount))
        Call item.ListSubItems.Add(, , Format(zeroTime(loopCount,_
        1), "##.###"))
    Next
    Call lstBounce.Refresh

    GoTo Exit_Form
Handle_Error:
    ' Error handling routine
    MsgBox (Err.Description)
Exit_Form:
End Sub
```

16

# How the MATLAB Builder NE Product Creates COM Components


- "Overview of Internal Processes" on page 16-2
- "Component Registration" on page 16-4
- "Data Conversion" on page 16-9
- "Calling Conventions" on page 16-23

# Overview of Internal Processes

**In this section...**

"How Is a MATLAB® Builder NE Component Created?" on page 16-2

"Code Generation" on page 16-2

"Create Interface Definitions" on page 16-3

"C++ Compilation" on page 16-3

"Linking and Resource Binding" on page 16-3

"Registration of the DLL" on page 16-3

## How Is a MATLAB Builder NE Component Created?

The process of creating a MATLAB Builder NE component is completely automatic from a user point of view. You specify a list of MATLAB files to process and a few additional pieces of information, such as the component name, the class names, and the version number.

## Code Generation

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

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

## Create Interface Definitions

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

## C++ Compilation

The third step compiles all C/C++ source files generated in steps 1 and 2 into object code. One additional file containing a set of C++ template classes (`mclcomclass.h`) is included at this point. This file contains template implementations of all necessary COM base classes, as well as error handling and registration code.

## Linking and Resource Binding

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

## Registration of the DLL

The final build step registers the DLL on the system, as described in "Component Registration" on page 16-4.

# Component Registration

**In this section...**

"Self-Registering Components" on page 16-4

"Globally Unique Identifier" on page 16-5

"Versioning" on page 16-7

## Self-Registering Components

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

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

```
mwregsvr mycomponent_1_0.dll
```

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

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

unregisters the component.

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

---

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

---

## Globally Unique Identifier

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

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

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

---

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

---

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

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

| Key | Definition |
|---|---|
| | *version-number*<br><br>These keys are useful for creating a class instance from the component and class names instead of the GUIDs. |

## Versioning

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

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

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

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

- If an existing component has the same version, it uses the GUID of the existing component's type library. If the name of the new class matches the

previous version, it reuses the class and interface GUIDs. If the class names do not match, it generates new GUIDs for the new class and interface.

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

# Data Conversion

**In this section...**


"Conversion Rules" on page 16-9

"Array Formatting Flags" on page 16-19

"Data Conversion Flags" on page 16-21


## Conversion Rules

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

---

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

---

---

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

---

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

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

The Win32 API provides many functions for creating and manipulating VARIANTs in C/C++, and Microsoft Visual Basic provides native language support for this type. See the Microsoft Visual Studio documentation for definitions and API support for COM VARIANTs. VARIANT variables are self describing and store their type code as an internal field of the structure.

---

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

---

See VARIANT Type Codes Supported on page 16-10 for a list of the VARIANT type codes supported by the builder components.

See MATLAB® to COM VARIANT Conversion Rules on page 16-12 and COM VARIANT to MATLAB® Conversion Rules on page 16-17 for conversion rules between COM VARIANTs and MATLAB arrays.

### VARIANT Type Codes Supported

| VARIANT Type Code (C/C++) | C/C++ Type | Variant Type Code (Visual Basic) | Visual Basic Type | Definition |
|---|---|---|---|---|
| VT_EMPTY | - | vbEmpty | - | Uninitialized VARIANT |
| VT_I1 | char | - | - | Signed one-byte character |
| VT_UI1 | unsigned char | vbByte | Byte | Unsigned one-byte character |
| VT_I2 | short | vbInteger | Integer | Signed two-byte integer |
| VT_UI2 | unsigned short | - | - | Unsigned two-byte integer |
| VT_I4 | long | vbLong | Long | Signed four-byte integer |

**VARIANT Type Codes Supported (Continued)**

| VARIANT Type Code (C/C++) | C/C++ Type | Variant Type Code (Visual Basic) | Visual Basic Type | Definition |
|---|---|---|---|---|
| VT_UI4 | unsigned long | - | - | Unsigned four-byte integer |
| VT_R4 | float | vbSingle | Single | IEEE® four-byte floating-point value |
| VT_R8 | double | vbDouble | Double | IEEE eight-byte floating-point value |
| VT_CY | CY+ | vbCurrency | Currency | Currency value (64-bit integer, scaled by 10,000) |
| VT_BSTR | BSTR+ | vbString | String | String value |
| VT_ERROR | SCODE+ | vbError | - | HRESULT (signed four-byte integer representing a COM error code) |
| VT_DATE | DATE+ | vbDate | Date | Eight-byte floating-point value representing date and time |
| VT_INT | int | - | - | Signed integer; equivalent to type int |
| VT_UINT | unsigned int | - | - | Unsigned integer; equivalent to type unsigned int |
| VT_DECIMAL | DECIMAL+ | vbDecimal | - | 96-bit (12-byte) unsigned integer, scaled by a variable power of 10 |

**VARIANT Type Codes Supported (Continued)**

| VARIANT Type Code (C/C++) | C/C++ Type | Variant Type Code (Visual Basic) | Visual Basic Type | Definition |
|---|---|---|---|---|
| VT_BOOL | VARIANT_BOOL+ | vbBoolean | Boolean | Two-byte Boolean value (0xFFFF = True; 0x0000 = False) |
| VT_DISPATCH | IDispatch* | vbObject | Object | IDispatch* pointer to an object |
| VT_VARIANT | VARIANT+ | vbVariant | Variant | VARIANT (can only be specified if combined with VT_BYREF or VT_ARRAY) |
| <*anything*>\|VT_ARRAY | | | | Bitwise combine VT_ARRAY with any basic type to declare as an array |
| <*anything*>\|VT_BYREF | | | | Bitwise combine VT_BYREF with any basic type to declare as a reference to a value |

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

**MATLAB to COM VARIANT Conversion Rules**

| MATLAB Data Type | VARIANT Type for Scalar Data | VARIANT Type for Array Data | Comments |
|---|---|---|---|
| cell | A 1-by-1 cell array converts to a single VARIANT with a type conforming to the conversion rule for the MATLAB data type of the cell contents. | A multidimensional cell array converts to a VARIANT of type VT_VARIANT\|VT_ARRAY with the type of each array member conforming to the | |

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

| MATLAB Data Type | VARIANT Type for Scalar Data | VARIANT Type for Array Data | Comments |
|---|---|---|---|
| | | conversion rule for the MATLAB data type of the corresponding cell. | |
| `structure` | `VT_DISPATCH` | `VT_DISPATCH` | A MATLAB `struct` array is converted to an `MWStruct` object. (See "Class MWStruct" on page 17-16.) This object is passed as a `VT_DISPATCH` type. |
| `char` | A 1-by-1 `char` matrix converts to a `VARIANT` of type `VT_BSTR` with string length = 1. | A 1-by-L `char` matrix is assumed to represent a string of length Lin MATLAB. This case converts to a `VARIANT` of type `VT_BSTR` with a string length = L. `char` matrices of more than one row, or of a higher dimensionality convert to a `VARIANT` 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. |

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

| MATLAB Data Type | VARIANT Type for Scalar Data | VARIANT Type for Array Data | Comments |
|---|---|---|---|
| sparse | VT_DISPATCH | VT_DISPATCH | A MATLAB sparse array is converted to an MWSparse object. (See "Class MWSparse" on page 17-26.) This object is passed as a VT_DISPATCH type. |
| double | A real 1-by-1 double matrix converts to a VARIANT of type VT_R8. A complex 1-by-1 double matrix converts to a VARIANT of type VT_DISPATCH. | A real multidimensional double matrix converts to a VARIANT of type VT_R8\|VT_ARRAY. A complex multidimensional double matrix converts to a VARIANT of type VT_DISPATCH. | Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. See "Class MWComplex" on page 17-24 |
| 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. |

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

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

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

| MATLAB Data Type | VARIANT Type for Scalar Data | VARIANT Type for Array Data | Comments |
|---|---|---|---|
| uint16 | A real 1-by-1 uint16 matrix converts to a VARIANT of type VT_UI2. A complex 1-by-1 uint16 matrix converts to a VARIANT of type VT_DISPATCH. | A real multidimensional uint16 matrix converts to a VARIANT of type VT_UI2\|VT_ARRAY. A complex multidimensional uint16 matrix converts to a VARIANT of type VT_DISPATCH. | Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. |
| int32 | A 1-by-1 int32 matrix converts to a VARIANT of type VT_I4. A complex 1-by-1 int32 matrix converts to a VARIANT of type VT_DISPATCH. | A multidimensional int32 matrix converts to a VARIANT of type VT_I4\|VT_ARRAY. A complex multidimensional int32 matrix converts to a VARIANT of type VT_DISPATCH. | Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. |
| uint32 | A 1-by-1 uint32 matrix converts to a VARIANT of type VT_UI4. A complex 1-by-1 uint32 matrix converts to a VARIANT of type VT_DISPATCH. | A multidimensional uint32 matrix converts to a VARIANT of type VT_UI4\|VT_ARRAY. A complex multidimensional uint32 matrix converts to a VARIANT of type VT_DISPATCH. | Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class. |
| Function handle | VT_EMPTY | VT_EMPTY | Not supported |
| Java class | VT_EMPTY | VT_EMPTY | Not supported |
| User class | VT_EMPTY | VT_EMPTY | Not supported |
| logical | VT_Bool | VT_Bool\|VT_ARRAY | |

### COM VARIANT to MATLAB Conversion Rules

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

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

| VARIANT Type | MATLAB Data Type (Scalar or Array Data) | Comments |
|---|---|---|
| VT_DATE | double | VARIANT dates are stored as doubles starting at midnight Dec. 31, 1899. MATLAB dates are stored as doubles starting at 0/0/00 00:00:00. Therefore, a VARIANT date of 0.0 maps to a MATLAB numeric date of 693960.0. VARIANT dates are converted to MATLAB double types and incremented by 693960.0.<br><br>VARIANT dates can be optionally converted to strings. See "Data Conversion Flags" on page 16-21 for more information on type coercion. |
| VT_INT | int32 | |
| VT_UINT | uint32 | |
| VT_DECIMAL | double | |
| VT_BOOL | logical | |
| VT_DISPATCH | *Varies* | IDispatch* pointers are treated within the context of what they point to. Objects must be supported types with known data extraction and conversion rules, or expose a generic Value property that points to a single VARIANT type. Data extracted from an object is converted based on the rules for the particular VARIANT obtained. |

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

| VARIANT Type | MATLAB Data Type (Scalar or Array Data) | Comments |
|---|---|---|
| | | Currently, support exists for Excel `Range` objects as well as the builder types `MWStruct`, `MWComplex`, `MWSparse`, and `MWArg`. See "Utility Library Classes" on page 17-3 for information on the builder types to use with COM components. |
| `anything|VT_BYREF` | *Varies* | Pointers to any of the basic types are processed according to the rules for what they point to. The resulting MATLAB array contains a deep copy of the values. |
| `anything|VT_ARRAY` | *Varies* | Multidimensional `VARIANT` arrays convert to multidimensional MATLAB arrays, each element converted according to the rules for the basic types. Multidimensional `VARIANT` arrays of type `VT_VARIANT|VT_ARRAY` convert to multidimensional cell arrays, each cell converted according to the rules for that specific type. |

## Array Formatting Flags

The builder components have flags that control how array data is formatted in both directions. Generally, you should develop client code that matches the intended inputs and outputs of the MATLAB functions with the corresponding methods on the compiled COM objects, in accordance with the rules listed in MATLAB® to COM VARIANT Conversion Rules on page 16-12 and COM

VARIANT to MATLAB® Conversion Rules on page 16-17. In some cases this is not possible, for example, when existing MATLAB code is used in conjunction with a third-party product like Excel.

The following table shows the array formatting flags.

**Array Formatting Flags**

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

**Array Formatting Flags (Continued)**

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

## Data Conversion Flags

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


- “CoerceNumericToType” on page 16-22
- “InputDateFormat” on page 16-22
- “OutputAsDate As Boolean” on page 16-22
- “DateBias As Long” on page 16-22

### CoerceNumericToType

This flag tells the data converter to convert all numeric VARIANT data to one specific MATLAB type. VARIANT type codes affected by this flag are VT_I1, VT_UI1, VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R8, VT_CY, VT_DECIMAL, VT_INT, VT_UINT, VT_ERROR, VT_BOOL, and VT_DATE. Valid values for this flag are mwTypeDefault, mwTypeChar, mwTypeDouble, mwTypeSingle, mwTypeLogical, mwTypeInt8, mwTypeUint8, mwTypeInt16, mwTypeUint16, mwTypeInt32, and mwTypeUint32.

The default for this flag, mwTypeDefault, converts numeric data according to the rules listed in "Data Conversion" on page 16-9.

### InputDateFormat

This flag tells the data converter how to convert VARIANT dates to MATLAB dates. Valid values for this flag are mwDateFormatNumeric (default) and mwDateFormatString. The default converts VARIANT dates according to the rule listed in VARIANT Type Codes Supported on page 16-10 . The mwDateFormatString flag converts a VARIANT date to its string representation. This flag only affects VARIANT type code VT_DATE.

### OutputAsDate As Boolean

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

### DateBias As Long

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

# Calling Conventions

**In this section...**


"Producing a COM Class" on page 16-23

"IDL Mapping" on page 16-24

"Microsoft® Visual Basic Mapping" on page 16-25


## Producing a COM Class

Producing a COM class requires the generation of

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

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

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

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

<table>
<tr><th>Code</th><th>Sample</th></tr>
<tr><td>Generic MATLAB Code</td><td><pre>function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)</pre></td></tr>
<tr><td>IDL Code</td><td><pre>HRESULT foo([in] long nargout,
            [in,out] VARIANT* Y1,
            [in,out] VARIANT* Y2,
            .
            .
            [in,out] VARIANT* varargout,
            [in] VARIANT X1,
            [in] VARIANT X2,
            .
            .
            [in] VARIANT varargin);</pre></td></tr>
<tr><td>Visual Basic Code</td><td><pre>Sub foo(nargout As Long, _
        Y1 As Variant, _
        Y2 As Variant, _
        .
        .
        varargout As Variant, _
        X1 As Variant, _
        X2 As Variant, _
        .
        .
        varargin As Variant)</pre></td></tr>
</table>

## IDL Mapping

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

When present, the `nargout` parameter is an `[in]` parameter of type `long`. It is always the first argument in the list. This parameter allows correct passage of the MATLAB `nargout` parameter to the compiled MATLAB code. The

nargout parameter is not produced if you encapsulate an MATLAB function containing no outputs.

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

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

When present, the optional varargin/varargout parameters are always listed as the last input parameters and the last output parameters. All parameters other than nargout are passed as COM VARIANT types. "Data Conversion" on page 16-9 lists the rules for conversion between MATLAB arrays and COM VARIANTs.

## Microsoft Visual Basic Mapping

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

MATLAB Builder NE components also provide direct support for the Microsoft Excel Range object, used by Visual Basic for Applications to represent a range of cells in an Excel worksheet.

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

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

# Utility Library for Microsoft COM Components


- “Referencing Utility Classes” on page 17-2
- “Utility Library Classes” on page 17-3
- “Enumerations” on page 17-31

# 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 (see “Utility Library Classes” on page 17-3) and three enumerated types (see “Enumerations” on page 17-31). Before using these types, you must make explicit references to the `MWComUtil` type libraries in the Microsoft Visual Basic IDE. To do this select **Tools > References** from the main menu of the 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.

---

# Utility Library Classes

**In this section...**


"Class MWUtil" on page 17-3

"Class MWFlags" on page 17-10

"Class MWStruct" on page 17-16

"Class MWField" on page 17-23

"Class MWComplex" on page 17-24

"Class MWSparse" on page 17-26

"Class MWArg" on page 17-29


## Class MWUtil

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


- "Sub MWInitApplication(pApp As Object)" on page 17-3
- "Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])" on page 17-5
- "Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])" on page 17-7
- "Sub MWDate2VariantDate(pVar)" on page 17-9


The function prototypes use Visual Basic syntax.

### Sub MWInitApplication(pApp As Object)

Initializes the library with the current instance of Microsoft Excel.

**Parameters.**

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

**Return Value.** None.

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

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

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

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

---

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

Packs a variable length list of `Variant` arguments into a single `Variant` array. This function is typically used for creating a `varargin` cell from a list of separate inputs. Each input in the list is added to the array only if it is nonempty and nonmissing. (In Visual Basic, a missing parameter is denoted by a `Variant` type of `vbError` with a value of `&H80020004`.)

**Parameters.**

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

**Return Value.** None.

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

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

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

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

```
Function mysum(Optional V0 As Variant, _
               Optional V1 As Variant, _
               Optional V2 As Variant, _
               Optional V3 As Variant, _
               Optional V4 As Variant, _
               Optional V5 As Variant, _
               Optional V6 As Variant, _
               Optional V7 As Variant, _
               Optional V8 As Variant, _
               Optional V9 As Variant) As Variant
Dim y As Variant
Dim varargin As Variant
Dim aClass As Object
Dim aUtil As Object

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

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

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

**Parameters.**

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

**Return Value.** None.

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

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

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

This function produces a sequence of nargout random column vectors, with the length of the ith vector equal to i. Assume that this function is included in a class named myclass that is included in a component named mycomponent with a version of 1.0. The Visual Basic subroutine takes no arguments and places the results into Excel columns starting at A1, B1, C1, and D1. If an error occurs, a message box displays the error text. This function assumes that MWInitApplication has been previously called.

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

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

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

### Sub MWDate2VariantDate(pVar)

Converts output dates from MATLAB to `Variant` dates.

**Parameters.**

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

**Return Value.** None.

**Remarks.** MATLAB handles dates as double-precision floating-point numbers with 0.0 representing 0/0/00 00:00:00. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as `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" on page 11-4 for more information on conversion between MATLAB and COM Automation types.) All MATLAB Builder for .NET COM components contain a reference to an MWFlags object that can modify data conversion rules at the object level. This class contains these properties and method:

- "Property ArrayFormatFlags As MWArrayFormatFlags" on page 17-10
- "Property DataConversionFlags As MWDataConversionFlags" on page 17-13
- "Sub Clone(ppFlags As MWFlags)" on page 17-15

### Property ArrayFormatFlags As MWArrayFormatFlags

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

- "Property InputArrayFormat As mwArrayFormat" on page 17-11

- "Property InputArrayIndFlag As Long" on page 17-12
- "Property OutputArrayFormat As mwArrayFormat" on page 17-12
- "Property OutputArrayIndFlag As Long" on page 17-13
- "Property AutoResizeOutput As Boolean" on page 17-13
- "Property TransposeOutput As Boolean" on page 17-13


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

**Array Formatting Rules for Input Arrays**

| Value | Behavior |
|---|---|
| `mwArrayFormatAsIs` | Converts arrays according to the default conversion rules listed in "Data Conversion Rules" on page 11-4. |
| `mwArrayFormatCell` | Coerces all arrays into cell arrays. Input scalar or numeric array arguments are converted to cell arrays with each cell containing a scalar value for the respective index. |
| `mwArrayFormatMatrix` | Coerces all arrays into matrices. When an input argument is encountered that is an array of `Variants` (the default behavior is to convert it to a cell array), the data converter converts this array to a matrix if each `Variant` is single valued, and all elements are homogeneous and of a numeric type. If this conversion is not possible, creates a cell array. |

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

**Property OutputArrayFormat As mwArrayFormat.** This property of type `mwArrayFormat` controls the formatting of arrays passed as output parameters to 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” on page 11-4. |
| `mwArrayFormatMatrix` | Coerces all arrays into matrices. When an output cell array argument is encountered (the default behavior converts it to an array of `Variants`), the data converter converts this array to a `Variant` that contains a simple numeric array if each cell is single valued, and all elements are homogeneous and of a numeric type. If this conversion is not possible, an array of `Variants` is created. |
| `mwArrayFormatCell` | Coerces all output arrays into arrays of `Variants`. Output scalar or numeric array arguments are converted to arrays of `Variants`, each `Variant` containing a scalar value for the respective index. |

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

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

**Property TransposeOutput As Boolean.** Setting this flag to `True` transposes the output arguments. This flag is useful when processing an output parameter from a method call on a COM component, where the MATLAB function returns outputs as row vectors, and you desire to place the data into columns. The default value for this flag is `False`.

## Property DataConversionFlags As MWDataConversionFlags

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

- "Property CoerceNumericToType As mwDataType" on page 17-13
- "Property InputDateFormat As mwDateFormat" on page 17-14
- "PropertyOutputAsDate As Boolean" on page 17-14
- "PropertyDateBias As Long" on page 17-14

**Property CoerceNumericToType As mwDataType.** This property converts all numeric input arguments to one specific MATLAB type. This flag is useful is when variables maintained within the Visual Basic code are different types, e.g., `Long`, `Integer`, etc., and all variables passed to the compiled MATLAB code must be doubles. The default value for this property is `mwTypeDefault`, which uses the default rules in "Data Conversion Rules" on page 11-4.

**Property InputDateFormat As mwDateFormat.** This property converts dates passed as input parameters to method calls on .NET Builder classes. The default value is `mwDateFormatNumeric`. The behaviors indicated by this flag are shown in the following table.

**Conversion Rules for Input Dates**

| Value | Behavior |
|---|---|
| `mwDateFormatNumeric` | Convert dates to numeric values as indicated by the rule listed in "Data Conversion Rules" on page 11-4. |
| `mwDateFormatString` | Convert input dates to strings. |

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

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

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

```
function p = myprimes(n)
if length(n)~=1, error('N must be a scalar'); end
if n < 2, p = zeros(1,0); return, end
p = 1:2:n;
q = length(p);
p(1) = 2;
for k = 3:2:sqrt(n)
    if p((k+1)/2)
```

```
        p(((k*k+1)/2):k:q) = 0;
    end
end
p = (p(p>0));
```

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

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

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

### Sub Clone(ppFlags As MWFlags)

Creates a copy of an MWFlags object.

**Parameters.**

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

**Return Value.** None

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

## Class MWStruct

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


- "Sub Initialize([varDims], [varFieldNames])" on page 17-16
- "Property Item([i0], [i1], ..., [i31]) As MWField" on page 17-18
- "Property NumberOfFields As Long" on page 17-21
- "Property NumberOfDims As Long" on page 17-21
- "Property Dims As Variant" on page 17-21
- "Property FieldNames As Variant" on page 17-21
- "Sub Clone(ppStruct As MWStruct)" on page 17-22


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

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

**Parameters.**

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

**Return Value.** None.

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

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

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

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

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

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

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

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

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

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

**Parameters.**

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


- "Property Name As String" on page 17-23
- "Property Value As Variant" on page 17-23
- "Property MWFlags As MWFlags" on page 17-23
- "Sub Clone(ppField As MWField)" on page 17-24


### Property Name As String

The name of the field (read only).

### Property Value As Variant

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

### Property MWFlags As MWFlags

Stores a reference to an MWFlags object. This property sets or gets the array formatting and data conversion flags for a particular field. Each field in a

structure has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

### Sub Clone(ppField As MWField)

Creates a copy of an `MWField` object.

**Parameters.**

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

**Return Value.** None.

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

## Class MWComplex

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


- "Property Real As Variant" on page 17-24
- "Property Imag As Variant" on page 17-25
- "Property MWFlags As MWFlags" on page 17-26
- "Sub Clone(ppComplex As MWComplex)" on page 17-26


### Property Real As Variant

Stores the real part of a complex array (read/write). The `Real` property is the default property of the `MWComplex` class. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to a numeric matrix (no cell data allowed).

Valid Visual Basic numeric types for complex arrays include `Byte`, `Integer`, `Long`, `Single`, `Double`, `Currency`, and `Variant/vbDecimal`.

### Property Imag As Variant

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

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

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

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

### Property MWFlags As MWFlags

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

### Sub Clone(ppComplex As MWComplex)

Creates a copy of an `MWComplex` object.

**Parameters.**

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

**Return Value.** None

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

## Class MWSparse

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


- "Property NumRows As Long" on page 17-27
- "Property NumColumns As Long" on page 17-27
- "Property RowIndex As Variant" on page 17-27
- "Property ColumnIndex As Variant" on page 17-27
- "Property Array As Variant" on page 17-27
- "Property MWFlags As MWFlags" on page 17-28
- "Sub Clone(ppSparse As MWSparse)" on page 17-28

### Property NumRows As Long

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

### Property NumColumns As Long

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

### Property RowIndex As Variant

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

### Property ColumnIndex As Variant

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

### Property Array As Variant

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

## Property MWFlags As MWFlags

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

## Sub Clone(ppSparse As MWSparse)

Creates a copy of an `MWSparse` object.

**Parameters.**

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

**Return Value.** None.

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

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

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

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

```
    Dim I As Long, K As Long

    On Error GoTo Handle_Error
    K = 1
    For I = 1 To 4
        rows(K) = I
        cols(K) = I + 1
        vals(K) = -1
        K = K + 1
        rows(K) = I
        cols(K) = I
        vals(K) = 2
        K = K + 1
        rows(K) = I + 1
        cols(K) = I
        vals(K) = -1
        K = K + 1
    Next
    rows(K) = 5
    cols(K) = 5
    vals(K) = 2
    Set x = New MWSparse
    x.NumRows = 5
    x.NumColumns = 5
    x.RowIndex = rows
    x.ColumnIndex = cols
    x.Array = vals
        .
        .
        .
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

## Class MWArg

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

- "Property Value As Variant" on page 17-30
- "Property MWFlags As MWFlags" on page 17-30
- "Sub Clone(ppArg As MWArg)" on page 17-30


### Property Value As Variant

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

### Property MWFlags As MWFlags

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

### Sub Clone(ppArg As MWArg)

Creates a copy of an `MWArg` object.

**Parameters.**

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

**Return Value.** None.

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

# Enumerations

**In this section...**


"Enum mwArrayFormat" on page 17-31

"Enum mwDataType" on page 17-31

"Enum mwDateFormat" on page 17-32


## Enum mwArrayFormat

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

### mwArrayFormat Values

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

## Enum mwDataType

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

### mwDataType Values

| Constant | Numeric Value | MATLAB Type |
|---|---|---|
| `mwTypeDefault` | 0 | Not applicable |
| `mwTypeLogical` | 3 | `logical` |
| `mwTypeChar` | 4 | `char` |
| `mwTypeDouble` | 6 | `double` |

**mwDataType Values (Continued)**

| Constant | Numeric Value | MATLAB Type |
|---|---|---|
| mwTypeSingle | 7 | single |
| mwTypeInt8 | 8 | int8 |
| mwTypeUint8 | 9 | uint8 |
| mwTypeInt16 | 10 | int16 |
| mwTypeUint16 | 11 | uint16 |
| mwTypeInt32 | 12 | int32 |
| mwTypeUint32 | 13 | uint32 |

## Enum mwDateFormat

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

**mwDateFormat Values**

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

# Examples

Use this list to find examples in the documentation.

## Magic Square Example for .NET

"Deploying a Component Using the Magic Square Example" on page 1-8

## Using Load and Save

"Using Load/Save Functions to Process MATLAB Data for Deployed Applications" on page 2-19

## Native Data Conversion

"Example: Native Data Conversion" on page 4-12

## Using functions engOpen and engEvalString from the MATLAB Engine API in a C# Program

"Example: Using functions engOpen and engEvalString from the MATLAB Engine API in a C# Program" on page 4-20

## Using WaitForFiguresToDie to Block Execution

"Code Fragment: Using WaitForFiguresToDie to Block Execution" on page 4-26

## Using the MCR Data Interface

"Improving Data Access Using the MCR User Data Interface and MATLAB® Builder NE" on page 4-33

## Sample Applications (C#)

"Simple Plot Example" on page 5-2
"Passing Variable Arguments" on page 5-7
"Spectral Analysis Example" on page 5-13
"Matrix Math Example" on page 5-20
"Phonebook Example" on page 5-28

## Sample Applications (Visual Basic .NET)

"Magic Square Example (Visual Basic)" on page 6-3
"Create Plot Example (Visual Basic)" on page 6-7
"Variable Arguments Example (Visual Basic)" on page 6-11
"Spectral Analysis Example (Visual Basic)" on page 6-15
"Matrix Math Example (Visual Basic)" on page 6-20
"Phonebook Example (Visual Basic)" on page 6-25

## Quick Start to Implementing a WebFigure

"Quick Start: Implementing a WebFigure" on page 7-7

## Working with Functions that Return a Single WebFigure as the Function's Only Output

"Working with Functions that Return a Single WebFigure as the Function's Only Output" on page 7-19

## Working With Functions That Return Multiple WebFigures In an Array as the Output

"Working With Functions That Return Multiple WebFigures In an Array as the Output" on page 7-20

## Attaching a WebFigure

"Attaching a WebFigure" on page 7-21

## Referencing a WebFigure Attached to the Local Server

"Referencing a WebFigure Attached to the Local Server" on page 7-26

## Referencing a WebFigure Attached to a Remote Server

"Referencing a WebFigure Attached to a Remote Server" on page 7-27

## Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up

"Using Global Application Class (Global.asax) to Create WebFigures at Server Start-Up" on page 7-28

## Creating and Modifying a MATLAB Figure

"Creating and Modifying a MATLAB Figure" on page 8-3

## Working with MATLAB Figures

"Working with Figures" on page 8-6

## Working with Images

"Working with Images" on page 8-7

## Building a Remotable Component Using the Deployment Tool

"Building a Remotable Component Using the Deployment Tool" on page 9-7

## Building a Remotable Component Using the mcc Command

"Building a Remotable Component Using the mcc Command" on page 9-10

## Using Native .NET Structure and Cell Arrays

"Coding and Building the Client Application and Configuration File with the Native MWArray, MWStructArray, and MWCellArray Classes " on page 9-34

## COM Components

"Building a Deployable COM Component" on page 13-2
"Packaging a Deployable COM Component" on page 13-4
"Calling a COM Object in a Visual C++ Program" on page 14-11
"Creating and Using a varargin Array in Microsoft® Visual Basic Programs" on page 14-26

"Creating and Using varargout in Microsoft® Visual Basic Programs" on page 14-27
"Using Array Formatting Flags" on page 14-30
"Using Data Conversion Flags" on page 14-33
"Blocking Execution of a Console Application that Creates Figures" on page 14-39
"Magic Square Example" on page 15-2
"Creating an Excel Add-In: Spectral Analysis Example" on page 15-10
"Univariate Interpolation Example" on page 15-25
"Matrix Calculator Example" on page 15-33
"Curve Fitting Example" on page 15-44
"Bouncing Ball Simulation Example" on page 15-52

## Utility Library Classes for COM Components

Chapter 17, "Utility Library for Microsoft COM Components"

# Glossary

**assembly**
Logical collection of one or more managed EXE or DLL files containing a .NET application's code and resources.

**CLS**
See Common Language Specification.

**common language runtime (CLR)**
Run-time environment provided by the .NET Framework, which runs the code and provides services that make the development process easier.

**Common Language Specification (CLS)**
A subset of language features supported by the **.NET common language runtime (CLR)**. CLS includes features common to several object-oriented programming languages, such as C#, VB.NET, and C++ with managed extensions. CLS-compliant components and tools are guaranteed to interoperate with other CLS-compliant components and tools.

**component installer**
The self-extracting executable created by the builder packaging process, which is used to deploy components created by the builder.

**.ctf files**
Component Technology Files, which are the encrypted MATLAB functions compiled by the builder. CTF archive files are embedded in the component generated by the builder as of R2008b.

**data conversion classes**
Provided by the builder to marshall data between MATLAB and other languages.

**feval API**
Interface generated by the builder for a MATLAB function. Includes both input and output arguments in the argument list. Output arguments are specified first, followed by the input arguments.

**finalization**

Semiautomatic mechanism provided by the .NET Framework to help clean up **native resources** just before garbage collection of a managed object.

**managed**

Code written in a programming language that uses the Microsoft .NET Framework. The languages share a unified set of class libraries and can be encoded into an Intermediate Language (IL). A CLR runtime-aware compiler compiles the IL into native executable code within a managed execution environment that ensures type safety, array bound and index checking, exception handling, and garbage collection.

**marshal**

To gather data from one or more applications and convert it to a format that is prescribed for a particular receiver or programming interface.

**MATLAB Compiler Runtime (MCR)**

Part of the MATLAB Builder NE product. Required to run MATLAB applications on machines that do not have the MATLAB desktop installed.

**mxArray**

The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, structures, and objects are stored as MATLAB arrays. In C, the MATLAB array is declared to be of type `mxArray`. The `mxArray` structure contains, among other things: its type, its dimensions, the data associated with this array, the number of fields and field names (if a structure or object).

**native code resources**

Resources that exist outside the control of the **CLR**.

**.NET Framework**

.NET is a software architecture developed by Microsoft to build component-based applications. .NET components expose interfaces that allow other managed applications and components to access their properties, methods, and events.

**Pascal case**

A convention for capitalizing identifier names. The first letter in the identifier and the first letter of each subsequent concatenated word is capitalized. For example: `MakeSquare`.

**project**

A feature of the MATLAB Builder NE product accessed via the Deployment Tool, which appears when you issue the `deploytool` command. A project specifies components and classes to be created and the functions to be associated with them.

**reflection**

Programming technique supported by **CLR**. Used to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object, and then invoke the type's methods or access its fields and properties.

**single output API**

Interface generated by MATLAB Builder NE when only a single output is required. Returns result as a single `MWArray` rather than an array of `MWArrays`.

**standard API**

Interface generated by MATLAB Builder NE. Specifies inputs within the argument list and outputs as an array of `MWArray` return values.

# Index


## A

access 14-2
Accessibility
  DLLs to add to path enabling 11-2
Advanced Encryption Standard (AES)
  cryptosystem 2-11
array formatting flags 14-29
Assistive technologies
  DLLs to add to path enabling 11-2

## B

build process 2-9

## C

capabilities 16-2
Class MWFlags 17-10
Class MWUtil 17-3
class name 4-8
class properties
  properties, class 14-36
COM
  defined 3-3
COM class
  producing 16-23
COM component
  as Excel add-in 15-10
  registration 16-4
  utility classes 17-1
  VB examples of creating and using 15-1
COM Components
  About 3-3
COM `VARIANT` 16-9
command line
  differences between command-line and
    GUI 2-8
command line interface 13-6
Command Line Interface 3-5
  Using .NET Bundles to Simply 3-5
Common Language Specification 3-2
Compiler
  security 2-11
compilers
  supported 11-2
component
  access 14-2
component indexing 4-22
Component Object Model (COM)
  defined 3-3
Component Technology File (CTF) 2-11
`componentinfo` function 12-2
Converting real or imaginary components
  MATLAB arrays and vectors
    ToArray 4-23
create phonebook example 5-28 6-25
`CreateObject` function 14-5
CTF (Component Technology File) archive 2-11
CTF Archive
  Controlling management and storage
    of. 4-32
CTF file 2-11

## D

data conversion
  classes for .NET components 11-7
  rules for .NET components 11-4
  rules for COM components 16-9
  utility classes for COM components 17-1
Data Conversion 4-9
data conversion flags 14-29
Data Structure Arrays
  Adding Fields 4-24
Data Structures
  Adding Fields 4-24
Data Types
  Casting in MATLAB® Builder NE 4-10
Dependency Analysis Function 2-9 to 2-10
depfun 2-9 to 2-10

Deployment Tool
differences between command-line and GUI 2-8
Starting from the command line 3-7 12-6
deploytool
differences between command-line and GUI 2-8
diagnostics 10-4
dispose 4-28
Distributing MATLAB code for sharing 1-15
DLLs 2-10
depfun 2-10
utility classes for COM components 17-1

**E**

Enumeration
mwArrayFormat 17-31
mwDataType 17-31
mwDateFormat 17-32
enumerations 17-31
error handling 4-27
errors 10-4
COM components 10-4
examples 5-28 6-25
C# 5-1
C# create plot 5-2
Excel add-in 15-10
magic square 15-2
Excel add-in 15-10
exceptions 4-27

**F**

Figures
Keeping open by blocking execution of console application 4-25
flags
array formatting 14-29
data conversion 14-29

**G**

Global Application Class (Global.asax) 7-28
global variables 14-36
Global.asax 7-28
Globally Unique Identifier (GUID) 16-5
GUID (Globally Unique Identifier) 16-5

**I**

IDL mapping 16-23

**L**

limitations 11-3
Limitations 3-3
Load function 2-19

**M**

magic square example 15-2
managed classes 4-7
MAT files 2-19
MATLAB Array indexing
About 4-24
MATLAB Builder NE
introduction 1-2
system requirements 11-2
MATLAB® Builder™ NE
example of deploying 1-8
MATLAB Compiler 11-2
build process 2-9
MATLAB Compiler Runtime (MCR)
defined 1-18 1-21 4-3
MATLAB Component Runtime (MCR)
Administrator Privileges, requirement of 1-22 4-3
Version Compatibility with MATLAB 1-22 4-3
MATLAB Data Conversion
Classes 4-9

MATLAB data files 2-19
MATLAB Data Types
  Automatic Casting in MATLAB® Builder NE 4-10
  Casting in MATLAB® Builder NE 4-10
MATLAB file
  encrypting 2-11
MATLAB Function Signatures
  Application Deployment product processing of 2-4
MATLAB objects
  no support for 3-3
MATLAB® Builder NE
  Building a Component 1-13
  Component and Class Naming Conventions 4-8
  Implementing Component On Another Computer 4-40
  Packaging a Component for distribution 1-15
  Using Classes and Methods with 4-7
  Using Component in .NET Application Coding 1-36
  Versioning 4-8
matrix math example
  C# 5-20
mcc
  differences between command-line and GUI 2-8
MCOS Objects
  no support for 3-3
MCR 1-23 4-4
  Sharing Among Classes 1-18
  singleton 1-18
MCR Component Cache
  How to use
    Overriding CTF embedding 4-32
messages 10-4
methods
  error handling 4-27
MEX-files 2-9 to 2-10
  depfun 2-10
missing parameter 17-5
multiple classes 5-13
MWArray class library 11-7
mwarray query
  return values 4-18
MWComponentOptions 4-32
`MWFlags` class 17-10
`mwregsvr` utility 16-4
`MWUtil` class 17-3

## N

Native .NET API
  Cell Arrays, Structs 9-27
  Data Structures 9-27
native resources
  dispose 4-28
*.NET common language runtime* (CLR) 3-2
.NET component
  C# examples of creating and using 5-1
  instantiating classes 1-30
  specifying 1-28
  VB examples of creating and using 6-1
.NET components
  overview of creating 1-8
`New` operator 14-6

## P

Packaging
  About 1-15
Parallel Computing Toolbox
  Configuring 4-33
  Example
    Using MATLAB Builder NE 4-34
  Supplying configuration information to 4-33
PCT
  Configuring 4-33
problems 10-4

## R

reflection 4-16
Remotable components 9-2
Renderers
    in WebFiigures 7-2
requirements
    system 11-2
restrictions 11-3
return values
    handling 4-16
    mwarray query 4-18
    reflection 4-16

## S

Save function 2-19
security 2-11
self-registering component 16-4
shared libraries 2-10
    depfun 2-10
shared library 2-10
Structs StructArrays
    Adding fields 4-24
system requirements 11-2

## T

troubleshooting 10-4
type library 16-4

## U

unregistering components 16-4
utility library 17-3

## V

`VARIANT` variable 16-9
version number
    components 16-7
versioning rules 16-7
Visual Basic mapping 16-25

## W

WaitForFiguresToDie 4-25
Web Figure
    WebFigure 7-2
WebFigures
    and MATLAB figures 2-6
    Getting image data from a WebFigure 8-8
    Returning a figure as data 2-6
    Returning data from a WebFigure
        Window 2-6
    Supported renderers 7-2