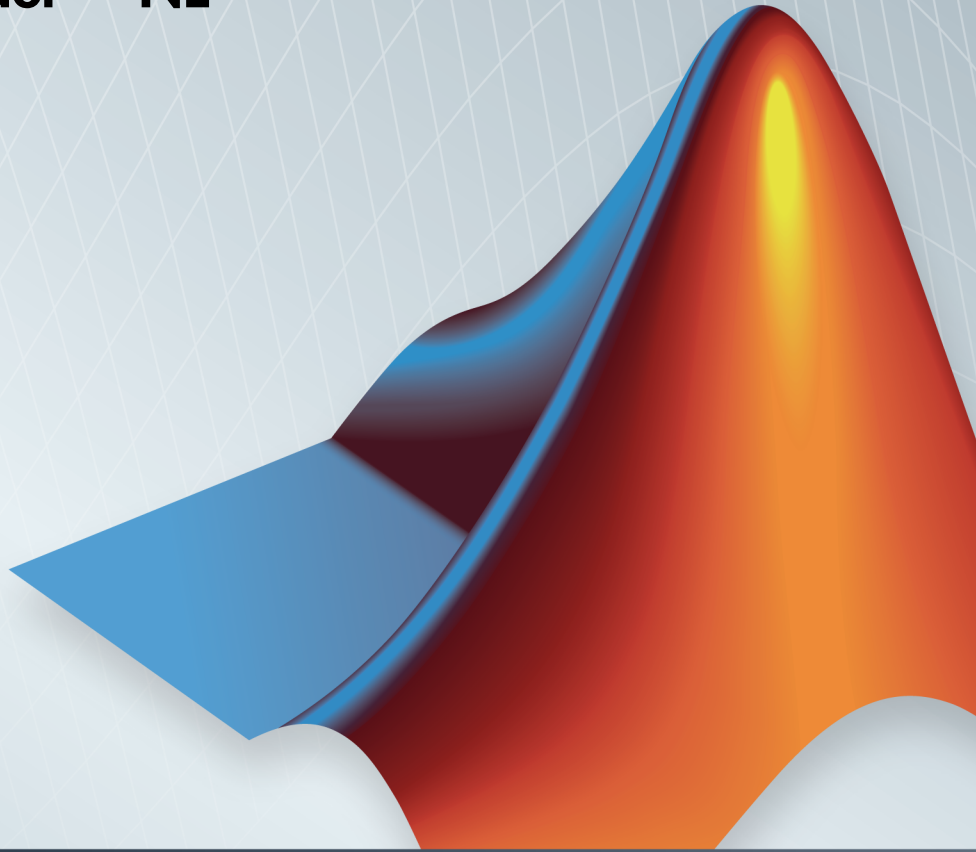


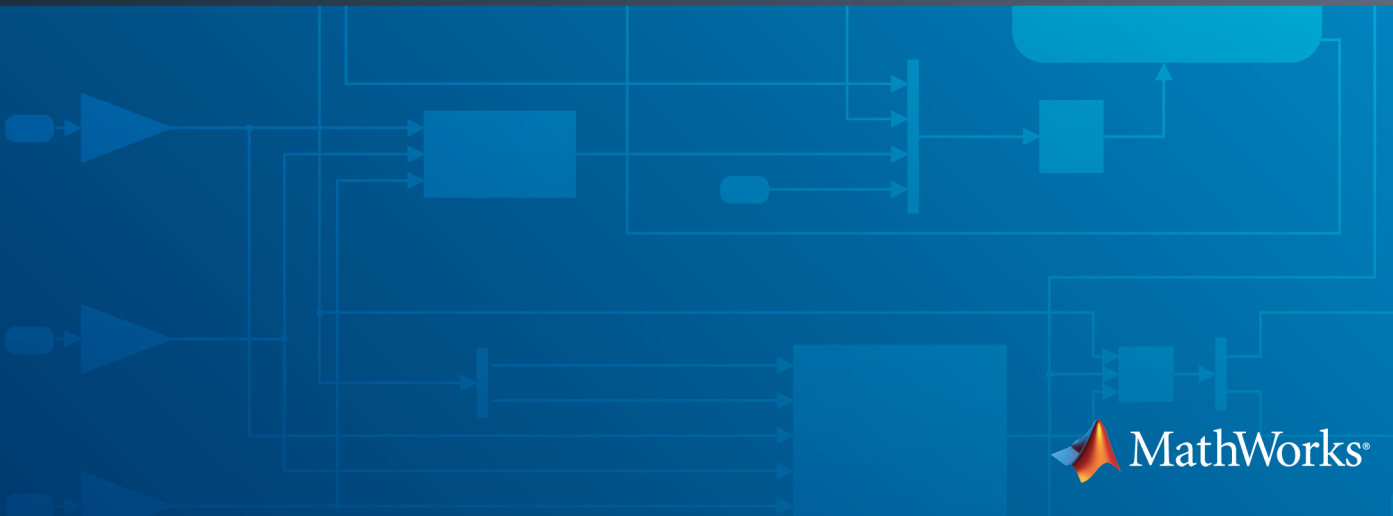
MATLAB[®] Builder[™] NE

User's Guide

R2014b



MATLAB[®]



How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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



Phone: 508-647-7000



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

MATLAB[®] Builder[™] NE User's Guide

© COPYRIGHT 2002–2014 by The MathWorks, Inc.

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

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks 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)
April 2011	Online only	Revised for Version 4.0 (Release 2011a)
September 2011	Online only	Revised for Version 4.1 (Release 2011b)
March 2012	Online only	Revised for Version 4.1.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.1.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.1.3 (Release 2013a)
September 2013	Online only	Revised for Version 4.2 (Release 2013b)
March 2014	Online only	Revised for Version 4.2.1 (Release R2014a)
October 2014	Online only	Revised for Version 4.2.2 (Release R2014b)

Getting Started

MATLAB Builder NE Product Description	1-2
Key Features	1-2
MATLAB Application Deployment Products	1-3
Appropriate Tasks for MATLAB Compiler and Builder Products	1-5
How MATLAB Builder NE Works	1-7
Roles in the .NET Application Deployment Process	1-8
MATLAB Builder NE Prerequisites	1-9
Products, Compilers, and IDE Installation	1-9
Deployment Target Architectures and Compatibility	1-10
MATLAB Builder NE Limitations	1-10
For More Information	1-10
Create a .NET Assembly From MATLAB Code	1-11
Integrate Your .NET Assembly Into a C# Application	1-17
Running the Component Installer	1-19
The Magic Square Component in an Enterprise C# Application	1-21

Application Deployment Products and the Compiler Apps .	2-2
What Is the Difference Between the Compiler Apps and the mcc Command Line?	2-2
How Does MATLAB Compiler Software Build My Application?	2-2
Dependency Analysis Function	2-5
MEX-Files, DLLs, or Shared Libraries	2-6
Deployable Archive	2-6
Write Deployable MATLAB Code	2-10
Compiled Applications Do Not Process MATLAB Files at Runtime	2-10
Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files	2-11
Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths	2-11
Gradually Refactor Applications That Depend on Noncompilable Functions	2-12
Do Not Create or Use Nonconstant Static State Variables	2-12
Get Proper Licenses for Toolbox Functionality You Want to Deploy	2-13
How the Deployment Products Process MATLAB Function Signatures	2-14
MATLAB Function Signature	2-14
MATLAB Programming Basics	2-14
Load MATLAB Libraries using loadlibrary	2-16
Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler	2-17
Use MATLAB Data Files (MAT Files) in Compiled Applications	2-18
Explicitly Including MAT files Using the %#function Pragma	2-18
Load and Save Functions	2-18
MATLAB Objects	2-21

3

Supported Compilation Targets	3-2
.NET Assembly	3-2
COM Components	3-3
Use the Library Compiler App to Build a .NET Assembly ...	3-4
Use the mcc Command Line to Build a .NET Assembly	3-5
Command-Line Syntax Description	3-5
Execute Compiler Projects with deploytool	3-6
Examples	3-8
For More Information	3-9

Customizing a Compiler Project

4

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

Common Integration Tasks	5-2
Application Coding	5-3
Using C# Code In an Integrated .NET Component	5-3
Data Conversion	5-4
MATLAB API Functions in a C# Program	5-16
Object Passing by Reference	5-17
Real or Imaginary Components Within Complex Arrays ...	5-20
Jagged Array Processing	5-22
Field Additions to Data Structures and Data Structure Arrays	5-22
MATLAB Array Indexing	5-22
Block Console Display When Creating Figures	5-23
Error Handling	5-25
Explicitly Freeing Resources With Dispose	5-26
C# Integration Examples	5-28
Simple Plot	5-28
Passing Variable Arguments	5-31
Spectral Analysis	5-36
Matrix Math	5-41
Phone Book	5-47
Optimization	5-52
Microsoft Visual Basic Integration Examples	5-58
Magic Square (Visual Basic)	5-58
Create Plot Example (Visual Basic)	5-61
Variable Arguments (Visual Basic)	5-63
Spectral Analysis (Visual Basic)	5-66
Matrix Math (Visual Basic)	5-69
Phone Book (Visual Basic)	5-73
Optimization (Visual Basic)	5-78
Component Access On Another Computer	5-83
For More Information	5-84

- Deploying Components to End Users** 6-2
 - Distributing MATLAB Code Using the MATLAB Runtime .. 6-2

- MATLAB Runtime Path Settings for Development and Testing** 6-5
 - Overview 6-5
 - Path for Java Development on All Platforms 6-5
 - Path Modifications Required for Accessibility 6-5
 - Windows Settings for Development and Testing 6-6
 - Linux Settings for Development and Testing 6-6
 - Mac Settings for Development and Testing 6-6

- MATLAB Runtime Path Settings for Run-time Deployment** . 6-7
 - General Path Guidelines 6-7
 - Path for Java Applications on All Platforms 6-7
 - Windows Path for Run-Time Deployment 6-7
 - Linux Paths for Run-Time Deployment 6-8
 - Mac Paths for Run-Time Deployment 6-8

- MATLAB Runtime Run-Time Options** 6-9
 - What Run-Time Options Can You Specify? 6-9
 - Getting MATLAB Runtime Option Values Using MWMCR .. 6-9

- MATLAB Runtime Component Cache and Deployable Archive Embedding** 6-11
 - Overriding Default Behavior 6-12
 - For More Information 6-12

- The MATLAB Runtime User Data Interface** 6-13
 - Supplying Cluster Profiles for Parallel Computing Toolbox Applications 6-13

- Impersonation Implementation Using ASP.NET** 6-18

- Enhanced XML Documentation Files** 6-22

7

Type-Safe Interfaces: An Alternative to Manual Data Marshaling	7-2
Advantages of Implementing a Type-Safe Interface	7-4
How Type-Safe Interfaces Work	7-5
Build Your Component and Generate Your Type-Safe API ..	7-8
Using the Library Compiler	7-8
Using the Command-Line Tools	7-8
Implementing a Type-Safe Interface	7-10
Data Conversion Rules for Using the Type-Safe Interface ..	7-11
Create Windows Communications Foundation (WCF)TM-Based Components	7-12
What Is WCF?	7-12
Before Running the WCF Example	7-13
Deploying a WCF-Based Component	7-13
Create Managed Extensibility Framework (MEF) Plug-Ins	7-25
What Is MEF?	7-25
MEF Prerequisites	7-26
Addition and Multiplication Applications with MEF	7-26

Web Deployment of Figures and Images

8

WebFigures	8-2
Supported Renderers for WebFigures	8-2
WebFigures Prerequisites	8-2
Quick Start Implementation of WebFigures	8-4
Advanced Configuration of a WebFigure	8-10
Upgrading Your WebFigures	8-23
Troubleshooting	8-24
Logging Levels	8-25

Create and Modify a MATLAB Figure	8-26
Preparing a MATLAB Figure for Export	8-26
Changing the Figure (Optional)	8-26
Exporting the Figure	8-27
Cleaning Up the Figure Window	8-27
Modify and Export Figure Data	8-27
Working with MATLAB Figure and Image Data	8-29
For More Comprehensive Examples	8-29
Work with Figures	8-29
Work with Images	8-30

.NET Remoting

9

What Is .NET Remoting?	9-2
What Are Remotable Components?	9-2
Benefits of Using .NET Remoting	9-2
Your Role in Building Distributed Applications	9-4
.NET Remoting Prerequisites	9-5
Select the Best Method of Accessing Your Component:	
MWArray API or Native .NET API	9-6
Using Native .NET Structure and Cell Arrays	9-7
Creating a Remotable .NET Component	9-8
Building a Remotable Component Using the Library Compiler App	9-8
Building a Remotable Component Using the mcc Command .	9-9
Files Generated by the Compilation Process	9-10
Enabling Access to a Remotable .NET Component	9-11
Using the MWArray API	9-11
Using the Native .NET API: Magic Square	9-16
Using the Native .NET API: Cell and Struct	9-22

10

Viewing the Latest Build Log	10-2
Failure to Find a Required File	10-3
Diagnostic Messages	10-4
Enhanced Error Diagnostics Using mstack Trace	10-7

Reference Information

11

Requirements for the MATLAB Builder NE Product	11-2
System and Compiler Requirements	11-2
Path Modifications Required for Accessibility	11-5
Data Conversion Rules	11-3
Managed Types to MATLAB Arrays	11-3
MATLAB Arrays to Managed Types	11-3
.NET Types to MATLAB Types	11-4
Character and String Conversion	11-9
Unsupported MATLAB Array Types	11-9
Overview of Data Conversion Classes	11-10
Overview	11-10
Returning Data from MATLAB to Managed Code	11-11
Example of MWNumericArray in a .NET Application	11-11
Interfaces Generated by the MATLAB Builder NE Product	11-11
MWArray Class Specification	11-16
Deployment Product Terms	11-17

12

Create and Install COM Components

13

Build a Deployable COM Component	13-2
Package a Deployable COM Component	13-3
Add-in and COM Component Registration	13-3
Embedded Deployable Archives	13-5
Use the Command-Line Interface	13-6
Install COM Components on a Target Computer	13-9

Programming with COM Components Created by the
MATLAB Builder NE Product

14

General Techniques	14-2
Registering and Referencing the Utility Library	14-3
Creating an Instance of a Class in Microsoft Visual Basic .	14-4
Advantages and Disadvantages	14-4
CreateObject Function	14-4
Microsoft Visual Basic New Operator	14-5
Advantages of Each Technique	14-5
Declaring a Reusable Class Instance	14-6
Calling the Methods of a Class Instance	14-7
Standard Mapping Technique	14-7
Variant	14-8
Examples of Passing Input and Output Parameters	14-8

Calling a COM Object in a Visual C++ Program	14-10
Using a COM Component in a .NET Application	14-11
Overview	14-11
Program Listings	14-11
Adding Events to COM Objects	14-12
MATLAB Language Pragma	14-12
Using a Callback with a Microsoft Visual Basic Event	14-13
Passing Arguments	14-16
Overview	14-16
Creating and Using a varargin Array in Microsoft Visual Basic Programs	14-16
Creating and Using varargout in Microsoft Visual Basic Programs	14-17
Passing an Empty varargin From Microsoft Visual Basic Code	14-17
Using Flags to Control Array Formatting and Data Conversion	14-19
Overview	14-19
Array Formatting Flags	14-19
Using Array Formatting Flags	14-20
Using Data Conversion Flags	14-22
Special Flags for Some Microsoft Visual Basic Types	14-24
Using MATLAB Global Variables in Microsoft Visual Basic	14-26
Blocking Execution of a Console Application That Creates Figures	14-29
MCRWaitForFigures	14-29
Using MCRWaitForFigures to Block Execution	14-29
MATLAB Runtime Run-Time Options	14-32
What MATLAB Runtime Options are Supported for COM?	14-32
How Do I Specify MATLAB Runtime Options?	14-32
Sharing a MATLAB Runtime Instance in COM or Java Applications	14-33
What Is a Singleton MATLAB Runtime?	14-33
Advantages and Disadvantages of Using a Singleton	14-33

Which Products Support Singleton MATLAB Runtime and How Do I Create a Singleton?	14-34
Obtaining Registry Information	14-35
Handling Errors During a Method Call	14-37

Using COM Components in Microsoft Visual Basic Applications

15

Magic Square Example	15-2
Example Overview	15-2
Creating the MATLAB File	15-2
Using the Library Compiler App to Create and Build the Project	15-2
Creating the Microsoft Visual Basic Project	15-3
Creating the User Interface	15-3
Creating the Executable in Microsoft Visual Basic	15-5
Testing the Application	15-5
Creating an Excel Add-in: Spectral Analysis Example	15-7
Example Overview	15-7
Building the Component	15-7
Integrating the Component with VBA	15-9
Creating the Microsoft Visual Basic Form	15-10
Adding the Spectral Analysis Menu Item to Microsoft Excel	15-15
Saving the Add-in	15-16
Testing the Add-in	15-16
Univariate Interpolation Example	15-19
Example Overview	15-19
Using the Library Compiler App to Create and Build the Component	15-19
Using the Component in Microsoft Visual Basic	15-19
Creating the Microsoft Visual Basic Form	15-20
Matrix Calculator Example	15-25
Example Overview	15-25
Building the Component	15-25

Using the Component in Microsoft Visual Basic	15-26
Creating the Microsoft Visual Basic Form	15-26
Curve Fitting Example	15-33
Example Overview	15-33
Building the Component	15-33
Using the Component in Microsoft Visual Basic	15-33
Creating the Microsoft Visual Basic Form	15-34
Bouncing Ball Simulation Example	15-39
Example Overview	15-39
Building the Component	15-39
Using the Component in Microsoft Visual Basic	15-40
Creating the Microsoft Visual Basic Form	15-40

How the MATLAB Builder NE Product Creates COM Components

16

Overview of Internal Processes	16-2
How Is a MATLAB Builder NE Component Created?	16-2
Code Generation	16-2
Create Interface Definitions	16-2
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	16-6
Data Conversion	16-8
Conversion Rules	16-8
Array Formatting Flags	16-17
Data Conversion Flags	16-18
Calling Conventions	16-20
Producing a COM Class	16-20
IDL Mapping	16-21

17

Utility Library for Microsoft COM Components

Referencing Utility Classes	17-2
Class MWUtil	17-3
Sub MWInitApplication(pApp As Object)	17-3
Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])	17-5
Function IsMCRJVMEEnabled() As Boolean	17-6
Function IsMCRInitialized() As Boolean	17-6
Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])	17-7
Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])	17-8
Sub MWDate2VariantDate(pVar)	17-10
Class MWFlags	17-12
Property ArrayFormatFlags As MWArrayFormatFlags ...	17-12
Property DataConversionFlags As MWDataConversionFlags	17-15
Sub Clone(ppFlags As MWFlags)	17-17
Class MWStruct	17-18
Sub Initialize([varDims], [varFieldNames])	17-18
Property Item([i0], [i1], ..., [i31]) As MWField	17-19
Property NumberOfFields As Long	17-22
Property NumberOfDims As Long	17-22
Property Dims As Variant	17-22
Property FieldNames As Variant	17-22
Sub Clone(ppStruct As MWStruct)	17-23
Class MWField	17-25
Property Name As String	17-25
Property Value As Variant	17-25
Property MWFlags As MWFlags	17-25
Sub Clone(ppField As MWField)	17-25

Class MWComplex	17-27
Property Real As Variant	17-27
Property Imag As Variant	17-27
Property MWFlags As MWFlags	17-28
Sub Clone(ppComplex As MWComplex)	17-28
Class MWSparse	17-30
Property NumRows As Long	17-30
Property NumColumns As Long	17-30
PropertyRowIndex As Variant	17-30
Property ColumnIndex As Variant	17-31
Property Array As Variant	17-31
Property MWFlags As MWFlags	17-31
Sub Clone(ppSparse As MWSparse)	17-31
Class MWArg	17-34
Property Value As Variant	17-34
Property MWFlags As MWFlags	17-34
Sub Clone(ppArg As MWArg)	17-34
Enum mwArrayFormat	17-36
Enum mwDataType	17-37
Enum mwDateFormat	17-38

Deploying .NET Components With the F# Programming Language

A

The Magic Square Example Using F#	A-2
Prerequisites	A-2
Step 1: Build the Component	A-2
Step 2: Integrate the Component Into an F# Application	A-2
Step 3: Deploy the Component	A-4

Getting Started

- “MATLAB Builder NE Product Description” on page 1-2
- “MATLAB Application Deployment Products ” on page 1-3
- “Appropriate Tasks for MATLAB Compiler and Builder Products” on page 1-5
- “How MATLAB Builder NE Works” on page 1-7
- “Roles in the .NET Application Deployment Process” on page 1-8
- “MATLAB Builder NE Prerequisites” on page 1-9
- “Create a .NET Assembly From MATLAB Code” on page 1-11
- “Integrate Your .NET Assembly Into a C# Application” on page 1-17
- “The Magic Square Component in an Enterprise C# Application” on page 1-21

MATLAB Builder NE Product Description

Deploy MATLAB code as .NET or COM components

MATLAB Builder NE lets you create .NET and COM components from MATLAB programs that include MATLAB math and graphics and user interfaces developed with MATLAB. You can integrate these components into larger .NET, COM, and Web applications and deploy them royalty-free to computers that do not have MATLAB installed using the MATLAB Compiler Runtime (MCR) that is provided with MATLAB Compiler™.

Using MATLAB Compiler, MATLAB Builder NE encrypts your MATLAB programs and then generates .NET or COM wrappers around them so that they can be accessed just like native .NET and COM components.

Key Features

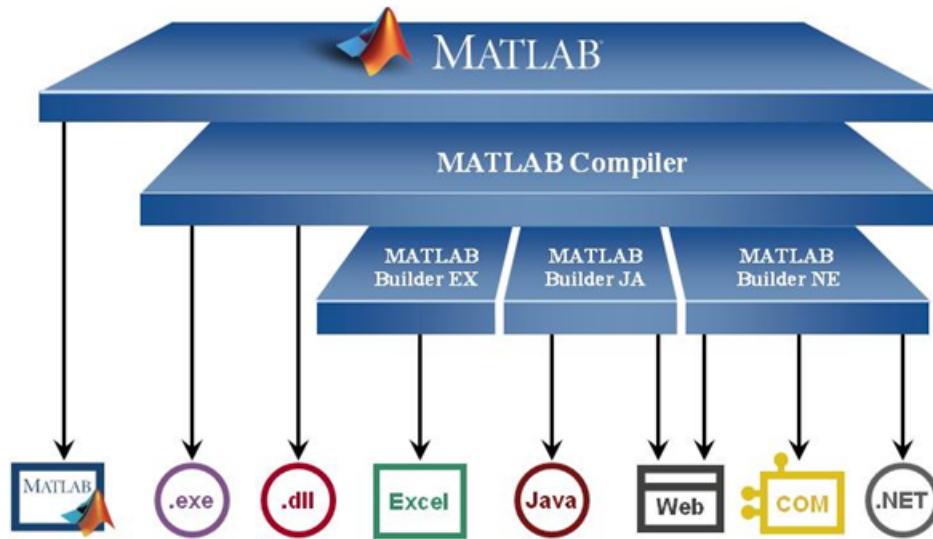
- Royalty-free desktop and Web deployment of .NET and COM components to computers that do not have MATLAB installed
- Components callable from Common Language Specification (CLS)-compliant languages, including C#, F#, VB.NET, or ASP.NET, and COM-compliant technology, including Visual Basic®, ASP, or Excel®
- Type-safe automatic conversion to and from native .NET, COM, and MATLAB data types
- Direct passing of .NET objects to and from a compiled MATLAB function
- Windows Communication Foundation (WCF) support with Web or enterprise service-oriented architecture (SOA)
- .NET remoting for interprocess communication
- WebFigures interface for MATLAB figure zooming, rotating, and panning

MATLAB Application Deployment Products

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

MATLAB Suite of Application Deployment Products

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



MATLAB Application Deployment Products

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

Appropriate Tasks for MATLAB Compiler and Builder Products

MATLAB Compiler compiles MATLAB code into standalone applications, libraries that can be integrated into other applications, or into deployable archives for use with MATLAB Production Server™. By default, MATLAB Compiler can generate standalone applications, C/C++ shared libraries, and deployable archives for use with MATLAB Production Server. Additional builders are available for Java, .NET, and Microsoft® Excel®.

While MATLAB Compiler lets you run your MATLAB application outside the MATLAB environment, it is not appropriate for all external tasks you may want to perform. Some tasks require either the MATLAB Coder™ product or MATLAB external interfaces. Use the following table to determine if MATLAB Compiler and builder products are appropriate to your needs.

MATLAB Compiler Task Matrix

Task	MATLAB Compiler and Builders	MATLAB Coder	MATLAB External Interfaces
Package MATLAB applications for deployment to users who do not have MATLAB	■		
Package MATLAB applications for deployment to MATLAB Production Server	■		
Build non-MATLAB applications that include MATLAB functions	■		
Generate readable, efficient, and embeddable C code from MATLAB code		■	
Generate MEX functions from MATLAB code for rapid prototyping and verification of generated C code within MATLAB		■	
Integrate MATLAB code into Simulink®		■	
Speed up fixed-point MATLAB code		■	

Task	MATLAB Compiler and Builders	MATLAB Coder	MATLAB External Interfaces
Generate hardware description language (HDL) from MATLAB code		■	
Integrate custom C code into MATLAB with MEX files			■
Call MATLAB from C and Fortran programs			■

For information on MATLAB Coder see “MATLAB Coder”.

For information on MATLAB external interfaces see “External Code Integration”.

How MATLAB Builder NE Works

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 runtime. For more information about the MATLAB runtime, see “Distributing MATLAB Code Using the MATLAB Runtime” on page 6-2.

For more information about how this product works with MATLAB Compiler, see “Write Deployable MATLAB Code” on page 2-10.

Roles 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
- Implement the infrastructure needed to successfully deploy a .NET application to the Web
- Create a remotable component that can be shared across distributed systems

The table Application Deployment Roles, Goals, and Tasks 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
MATLAB programmer	<ul style="list-style-type: none"> • MATLAB expert • No IT experience • No access to IT systems 	<ul style="list-style-type: none"> • Develops models; implements in MATLAB • Uses tools to create a component that is used by the .NET programmer
.NET developer	<ul style="list-style-type: none"> • Little or no MATLAB experience • Moderate IT experience • .NET expert • Minimal access to IT systems 	<ul style="list-style-type: none"> • Integrates deployed component with the rest of the .NET application • Integrates deployed MATLAB Figures with the rest of the .NET application

MATLAB Builder NE Prerequisites

In this section...

“Products, Compilers, and IDE Installation” on page 1-9

“Deployment Target Architectures and Compatibility” on page 1-10

“MATLAB Builder NE Limitations” on page 1-10

“For More Information” on page 1-10

Products, Compilers, and IDE Installation

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.

Microsoft .NET Framework Installation

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” for details.

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”

Deployment Target Architectures and Compatibility

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

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

MATLAB Builder NE Limitations

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.

Using `addAssembly` (External Interfaces)

.NET assemblies or DLLs built with MATLAB Builder NE cannot be loaded back into MATLAB with the NET External Interface method `addAssembly`.

Serialization of MATLAB Objects Unsupported

There is no support in MATLAB Builder NE for serializing MATLAB objects from MATLAB into .NET code.

For More Information

If you want to...	See...
Deploy a .NET assembly	“Create a .NET Assembly From MATLAB Code” on page 1-11
Deploy a COM component	“Magic Square Example”
Deploy a figure or image to the Web	“WebFigures” on page 8-2
Deploy a <i>remotable component</i>	“Creating a Remotable .NET Component” on page 9-8

Create a .NET Assembly From MATLAB Code

This example shows how to transform a MATLAB function into a .NET assembly. The example compiles a MATLAB function, `makesquare`, which computes a magic square into a .NET assembly named `MagicSquareComp`, which contains the `MLTestClass` class and other files needed to deploy your application.

In MATLAB, create the function that you want to deploy as a shared library.

This example uses the sample function, called `makesquare.m`, included in the `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MagicSquareExample\MagicSquareComp` folder, where `matlabroot` represents the name of your MATLAB installation folder.

```
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 2001-2007 The MathWorks, Inc.

y = magic(x);
```

Run the example in MATLAB.

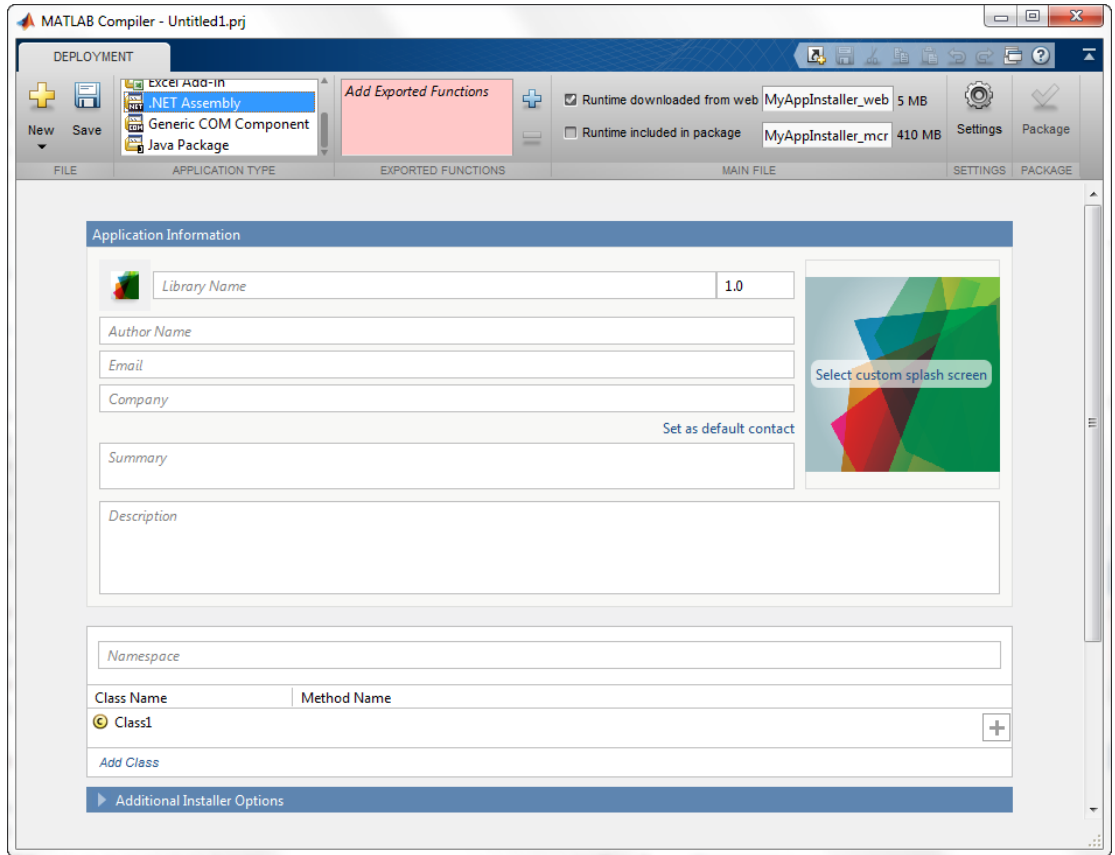
```
makesquare(5)
```

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

Open the **Library Compiler** app.

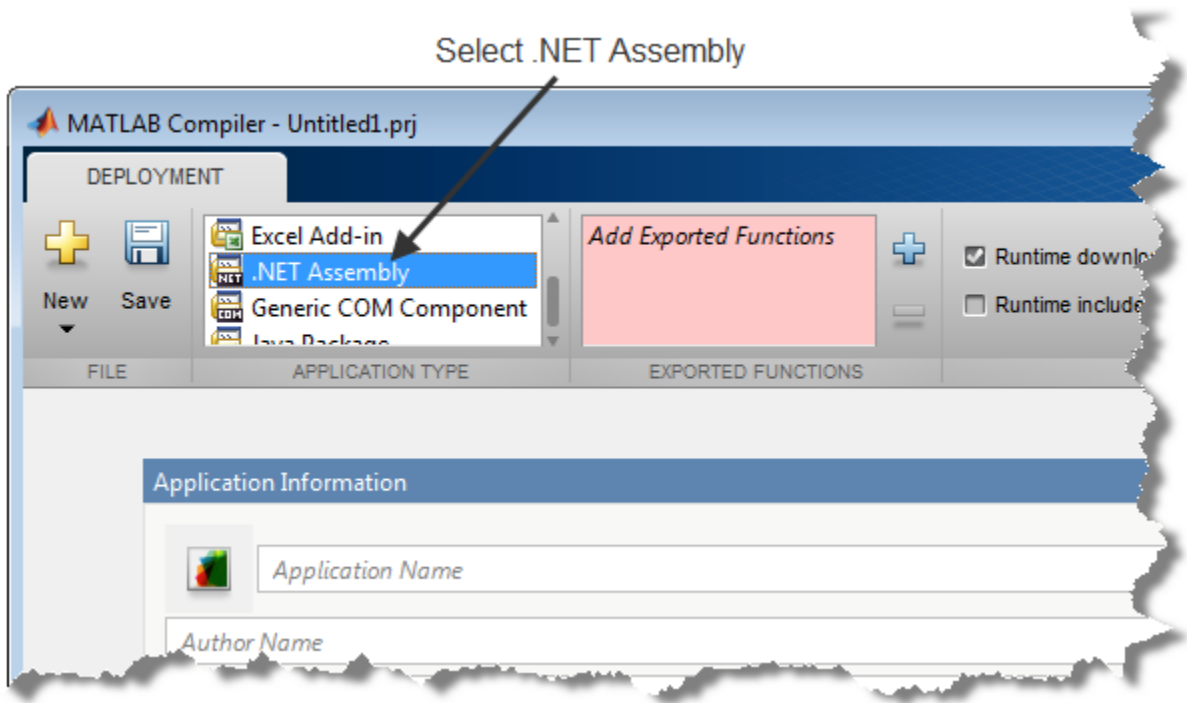
- a** On the toolstrip, select the **Apps** tab.
- b** Click the arrow at the far right of the tab to open the apps gallery.
- c** Click **Library Compiler**.

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



In the **Application Type** section of the toolbar, select **.NET Assembly** from the list.

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



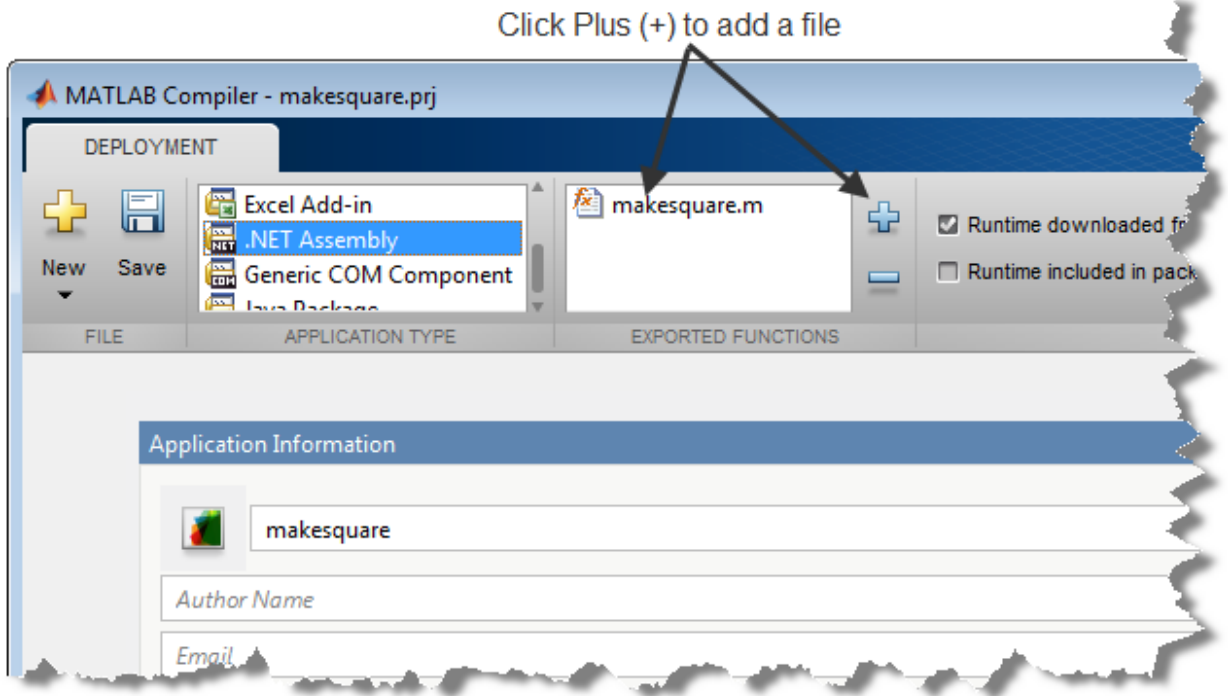
Specify the MATLAB functions that you want to deploy.

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

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

- b In the file explorer that opens, locate and select the `makesquare.m` file.
- c Click **Open** to select the file and close the file explorer.

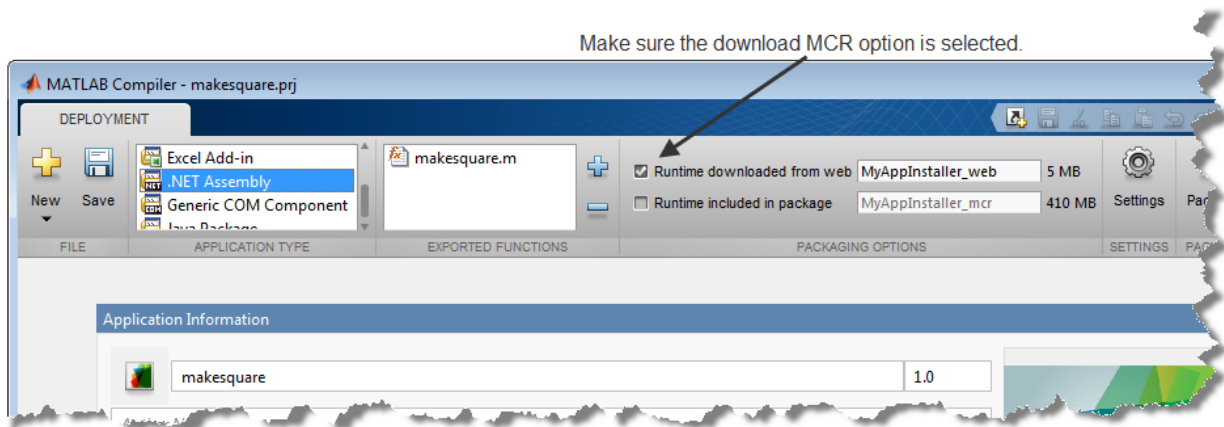
The Library Compiler adds the file you select (**makesquare.m**) to the list of files and a minus button appears under the plus button. The Library Compiler uses the name of the file as the name of the deployment project file (`.prj`), shown in the title bar, and as the name of the assembly, shown in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



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

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

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



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

The project window is divided into the following areas:

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

Click **Package**.

The Package window opens while the library is being generated. Select the **Open output folder when process completes** check box. The packaging process generates a self-extracting file that *automatically registers* the DLL and unpacks all deployable deliverables.

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

It should contain:

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

Click **Close** on the Package window.

Integrate Your .NET Assembly Into a C# Application

This example shows how to call a .NET assembly built with MATLAB Builder NE from a C# application.

- 1 Install the .NET assembly and the MATLAB runtime.

The creator of the assembly makes these files available by distributing the installer created by MATLAB Builder NE during the build process. The installer is located in the `for_redistribution` folder of your deployment project. The installer places the .NET assembly on your computer and automatically installs the MATLAB runtime from the Web, if you do not already have the MATLAB runtime installed on your system.

You can also download the MATLAB runtime installer from <http://www.mathworks.com/products/compiler/mcr>. The generated shared libraries and support files are located in the MATLAB deployment project's `for_testing` folder.

- 2 Open Microsoft Visual Studio
- 3 Create a new project.

For this example create a C# Console Application called **MainApp**.

- 4 Create a reference to your assembly.

The assembly is located in the application folder created when you installed the component. For this example, select `makeSqr.dll`.

- 5 Create a reference to the `MWArray` API.

The API is located in `MATLABROOT\toolbox\dotnetbuilder\bin\arch\version\MWArray.dll`.

- 6 Make .NET Namespaces available to your generated component and `MWArray` libraries. Add the following `using` statements to your C#.NET code:

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

In this example name the component `demo`.

- 7 Initialize your classes before you use them.

```
namespace mainApp  
{  
    class Program
```

```
{
    static void Main(string[] args)
    {
        MLTestClass obj = null;
        MWNumericArray input = null;
        MWNumericArray output = null;
        MWArray[] result = null;
    }
}
```

- 8** Instantiate your component class.

```
obj = new MLTestClass();
```

- 9** Invoke your component. After you complete the tasks of initializing and instantiating the classes you are working with, invoke the `makeSqr` component. 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.

When calling your component, you can take advantage of implicit conversion from .NET types to MATLAB types, by passing the native C# value directly to `makeSqr`:

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

You can also use explicit conversion:

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

- 10** 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);
```

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

```
using System;
using System.Collection.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;

            try
            {
                obj = new MLTestClass();

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

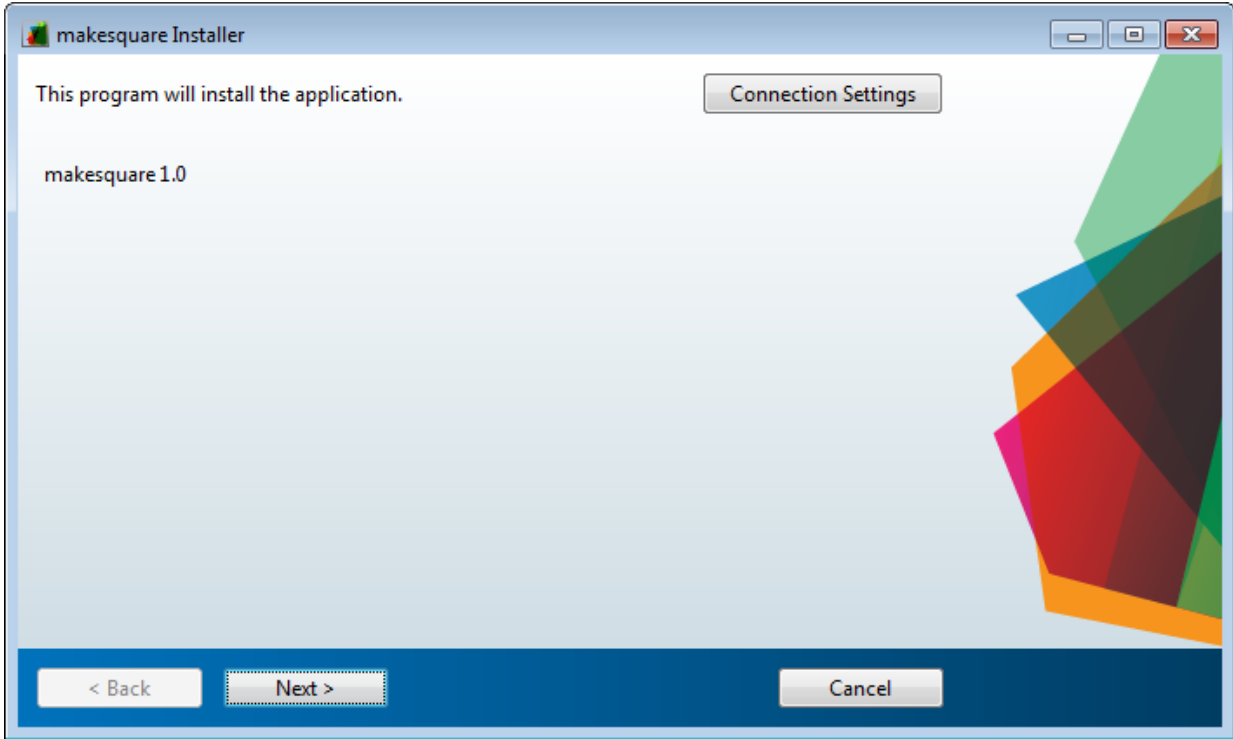
                output = (MWNumericArray)result[0];
                Console.WriteLine(output);
            }
            catch
            {
                throw;
            }
        }
    }
}
```

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

Running the Component Installer

MATLAB Builder NE creates an installer for the generated .NET component. After compilation is complete, you can find this installer in the `for_redistribution` folder in your project folder. By default, the compiler names the installer `MyAppInstaller_web.exe` or `MyAppInstaller_mcr.exe`, depending on which packaging option you chose. Using the Application Information area of the Library Compiler app, you can customize the look of the installer.

For example, when an end-user double-clicks the component installer, the first screen identifies you component by name and version number.



By clicking **Next** on each screen, the installer leads you through the installation process. During installation, you can specify the installation folder. The installer also automatically downloads the MATLAB runtime, if needed.

The Magic Square Component in an Enterprise C# Application

- 1 Write source code for an application that uses the .NET component created in “Create a .NET Assembly From MATLAB Code” on page 1-11.

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

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 C#, as well as Microsoft Visual Basic examples, see “C# Integration Examples” on page 5-28 and “Microsoft Visual Basic Integration Examples” on page 5-58.

- 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 Add a reference to the `MWArray` component in `matlabroot\toolbox\dotnetbuilder\bin\architecture\framework_version`.

See “Supported Microsoft .NET Framework Versions” 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.

MATLAB Code Deployment

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

Application Deployment Products and the Compiler Apps

In this section...

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

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

“Dependency Analysis Function” on page 2-5

“MEX-Files, DLLs, or Shared Libraries” on page 2-6

“Deployable Archive” on page 2-6

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

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

Compiler app advantages include:

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

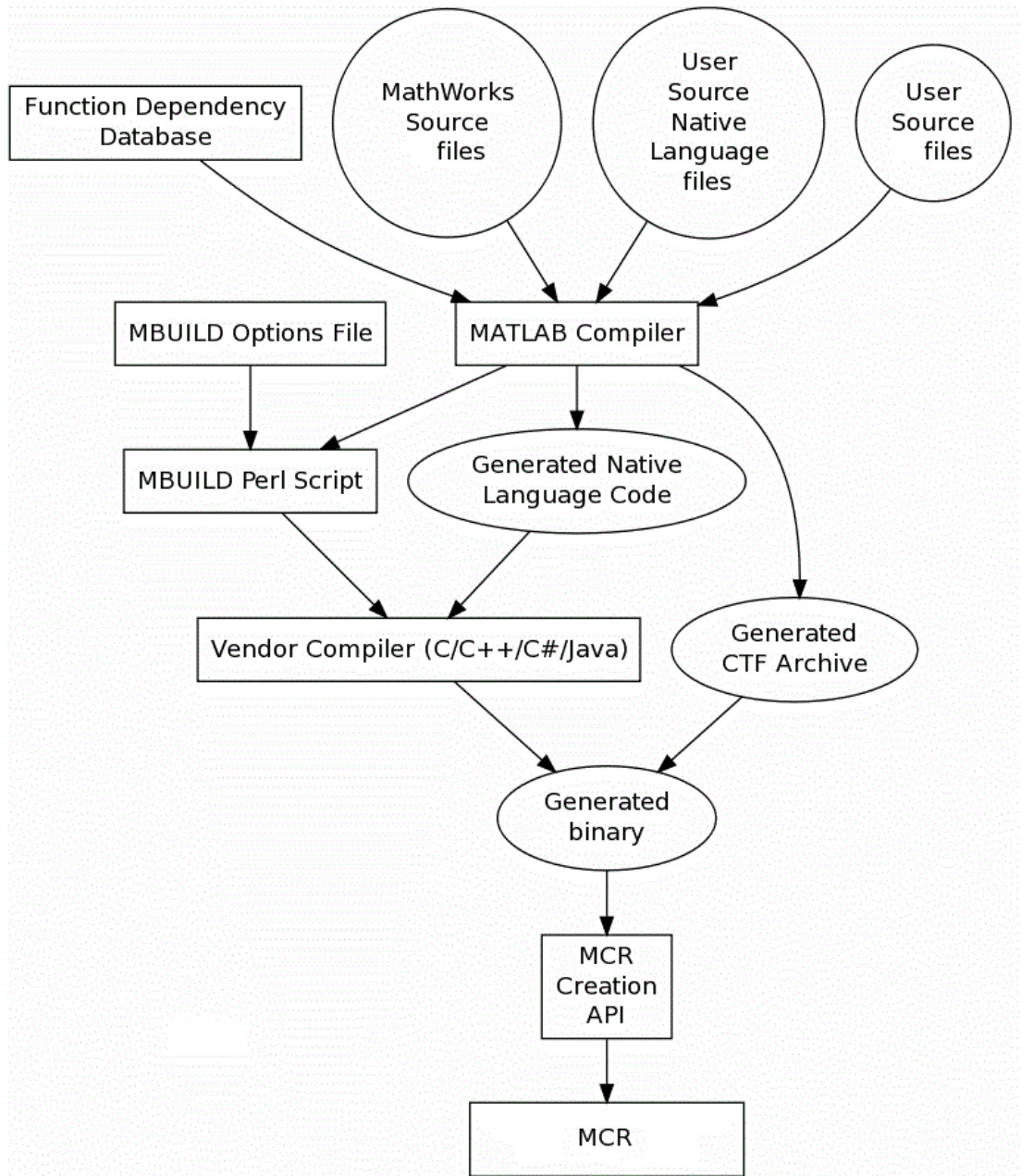
How Does MATLAB Compiler Software Build My Application?

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

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

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

For more information about how the compiler does dependency analysis, see “Dependency Analysis Function” on page 2-5.



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

- 3 Validates MEX-files. In particular, `mexFunction` entry points are verified. For more details about MEX-file processing, see “MEX-Files, DLLs, or Shared Libraries” on page 2-6.
- 4 Creates a deployable archive from the input files and their dependencies. For more details about deployable archives see “Deployable Archive” on page 2-6.
- 5 Generates target-specific wrapper code. For example, a C main function requires a very different wrapper than the wrapper for a Java interface class.
- 6 Generates target-specific binary package. For library targets such as C++ shared libraries, Java packages, or .NET assemblies, the compiler will invoke the required third-party compiler.

Dependency Analysis Function

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

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

The dependency analyzer searches for executable content such as:

- MATLAB files
- P-files

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

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

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

MEX-Files, DLLs, or Shared Libraries

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

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

Deployable Archive

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

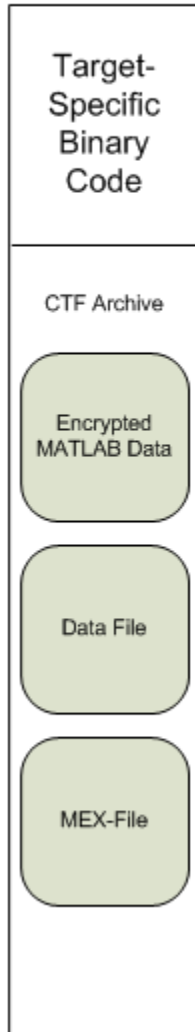
If you choose to extract the deployable archive as a separate file, the files remain encrypted. For more information on how to extract the deployable archive refer to the references in the following table.

Information on Deployable Archive Embedding/Extraction and Component Cache

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

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

Generated Component (EXE, DLL, SO, etc)



Additional Details

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

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

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

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

Write Deployable MATLAB Code

In this section...

“Compiled Applications Do Not Process MATLAB Files at Runtime” on page 2-10

“Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files” on page 2-11

“Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths” on page 2-11

“Gradually Refactor Applications That Depend on Noncompilable Functions” on page 2-12

“Do Not Create or Use Nonconstant Static State Variables” on page 2-12

“Get Proper Licenses for Toolbox Functionality You Want to Deploy” on page 2-13

Compiled Applications Do Not Process MATLAB Files at Runtime

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

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

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

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

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

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

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

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

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

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

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

If you find you cannot avoid placing `addpath` calls in your MATLAB code, use `ismcc` and `isdeployed`. See “Use `ismcc` and `isdeployed` Functions To Execute Deployment-Specific Code Paths” on page 2-11 for details.

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

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

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

Gradually Refactor Applications That Depend on Noncompilable Functions

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

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

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

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

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

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

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

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

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

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

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

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

Get Proper Licenses for Toolbox Functionality You Want to Deploy

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

How the Deployment Products Process MATLAB Function Signatures

In this section...
“MATLAB Function Signature” on page 2-14
“MATLAB Programming Basics” on page 2-14

MATLAB Function Signature

MATLAB supports multiple signatures for function calls.

The generic MATLAB function has the following structure:

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

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

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

All arguments represent a specific MATLAB type.

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

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

MATLAB Programming Basics

Creating a Deployable MATLAB Function

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

```
>> 1 + 1
```

cannot be deployed.

However, the following calculation:

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

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

Taking Inputs into a Function

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

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

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

Load MATLAB Libraries using loadlibrary

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

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

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

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

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

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

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

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

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

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

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

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

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

Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler

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

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

Use MATLAB Data Files (MAT Files) in Compiled Applications

In this section...
“Explicitly Including MAT files Using the <code>%#function</code> Pragma” on page 2-18
“Load and Save Functions” on page 2-18
“MATLAB Objects” on page 2-21

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

MATLAB Compiler excludes MAT files from “Dependency Analysis Function” on page 2-5 by default.

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

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

Load and Save Functions

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

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

For more information about deployable archives, see “Deployable Archive” on page 2-6.

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

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

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

The following example specifies three MATLAB data files:

- user_data.mat
- userdata\extra_data.mat
- ..\externdata\extern_data.mat

1 Navigate to *matlab_root*\extern\examples\compiler\Data_Handling.

2 Compile ex_loadsave.m with the following mcc command:

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

ex_loadsave.m

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

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

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

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

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

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

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

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

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

MATLAB Objects

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

```
%#function class_constructor
```

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

Component Building

- “Supported Compilation Targets” on page 3-2
- “Use the Library Compiler App to Build a .NET Assembly” on page 3-4
- “Use the mcc Command Line to Build a .NET Assembly” on page 3-5
- “Examples” on page 3-8
- “For More Information” on page 3-9

Supported Compilation Targets

In this section...
“.NET Assembly” on page 3-2
“COM Components” on page 3-3

.NET Assembly

MATLAB Builder NE supports compilation (building) of .NET assemblies 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.

What are .NET Assemblies and When Should You Create Them?

.NET assemblies are executable assemblies written in an CLS-supported language, such as C#, F#, or Microsoft Visual Basic. They execute in applications executing within the Microsoft .NET Framework.

.NET assemblies are special types of executable built from .NET project using Microsoft Visual Studio. After compilation, the assembly is referenced by applications that consume the services the component provides.

In many .NET Web environments, assemblies run on a Web server and provide data and other services (such as security, communications, and graphics) to Web Services—operating on the server.

.NET assemblies provide a programmable interface that is accessed by client applications. The assembly interface consists of properties, methods, and events that are exposed by the classes contained within the component. In other words, an assembly is a compiled set of classes that support the services provided by the component. The

classes expose their services through the properties, methods, and events that comprise the component's interface.

For development primarily within the Microsoft Windows® environment, .NET assemblies and applications provide scalable solutions for applications in large-scale enterprise or Web environments.

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 “Build a Deployable COM Component” on page 13-2, “Calling a COM Object in a Visual C++ Program” on page 14-10, and “Install COM Components on a Target Computer” on page 13-9.

Use the Library Compiler App to Build a .NET Assembly

For a complete example of how to build .NET assembly using the graphical interface, read “Create a .NET Assembly From MATLAB Code” on page 1-11 in this User's Guide.

For information about how the graphical interface differs from the command line interface, read “What Is the Difference Between the Compiler Apps and the mcc Command Line?”

Use the mcs Command Line to Build a .NET Assembly

In this section...

“Command-Line Syntax Description” on page 3-5

“Execute Compiler Projects with deploytool” on page 3-6

Command-Line Syntax Description

Instead of using the Library Compiler app to create .NET assemblies, you can use the `mcs` command.

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

```
mcs -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
```

.NET Bundle Files

You can simplify the command line used to create .NET assemblies. To do so, use the 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 assembly 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 bundle file.

```
mcs -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” for a list of supported framework versions.

Creating a .NET Namespace

The following example creates a .NET assembly 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 assembly 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” for a list of supported framework versions.

Adding Multiple Classes to an Assembly

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

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

See “Supported Microsoft .NET Framework Versions” for a list of supported framework versions.

Execute Compiler Projects with `deploytool`

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build project_name` — Invoke the compiler to build the project and not generate an installer.

- `-package project_name` — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsquare` generates the binary files defined by the `magicsquare` project and packages them into an installer that you can distribute to others.

Examples

See “C# Integration Examples” on page 5-28 and “Microsoft Visual Basic Integration Examples” on page 5-58 for complete examples of how to build and integrate .NET assemblies.

For More Information

If you want to...	See...
Learn how to build a assembly	“Create a .NET Assembly From MATLAB Code” on page 1-11
Learn how to perform basic integration tasks using C# code	“Integrate Your .NET Assembly Into a C# Application” on page 1-17
<ul style="list-style-type: none">• Basic MATLAB Programmer tasks• How the deployment products process your MATLAB functions• How the deployment products work together	“Write Deployable MATLAB Code” on page 2-10
Integration tasks for the .NET Developer	“Common Integration Tasks” on page 5-2
The MATLAB Component Runtime	“Distributing MATLAB Code Using the MATLAB Runtime” on page 6-2

Customizing a Compiler Project

- “Customize the Installer” on page 4-2
- “Manage Required Files in a Compiler Project” on page 4-6
- “Specify Files to Install with the Application” on page 4-8
- “Manage Support Packages” on page 4-9

Customize the Installer

In this section...

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

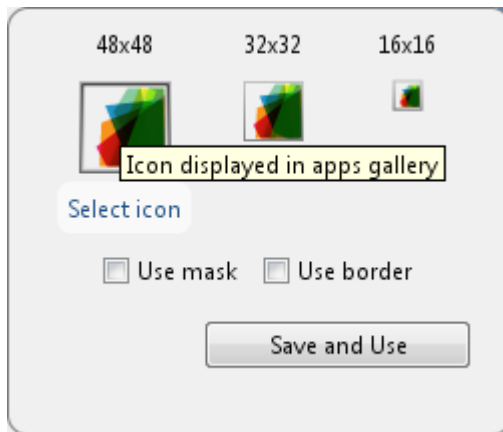
Change the Application Icon

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

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

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

A window previewing the icon opens.



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

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

Add Application Information

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

- Name

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

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

- Version

The default value is 1.0.

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

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

- Summary
- Description

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

Change the Splash Screen

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

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

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

Change the Installation Path

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

Default Installation Paths

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

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

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

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

Custom Installation Roots

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

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

Change the Logo

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

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

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

Edit the Installation Notes

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

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

Manage Required Files in a Compiler Project

In this section...
“Dependency Analysis” on page 4-6
“Using the Compiler Apps” on page 4-6
“Using mcc” on page 4-6

Dependency Analysis

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

Using the Compiler Apps

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

To add files:

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

To remove files:

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

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

Using mcc

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

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

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

Specify Files to Install with the Application

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

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

to add files to the list:

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

To remove files from the list:

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

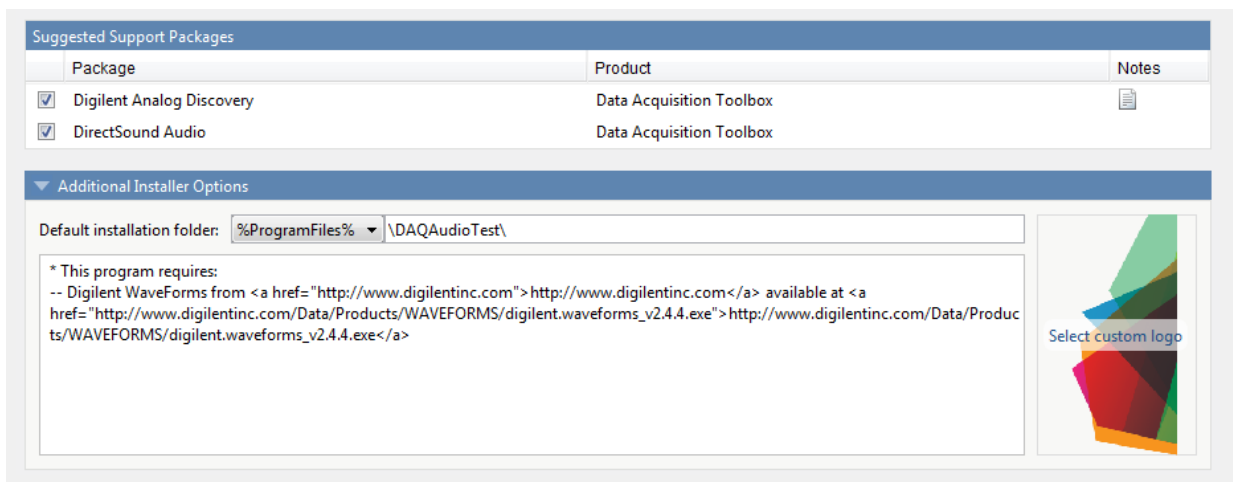
Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

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

Manage Support Packages

Using a Compiler App

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



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

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

Deselect support packages that are not required by your application.

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

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

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

Using the Command Line

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

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

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

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

Component Integration

- “Common Integration Tasks” on page 5-2
- “Application Coding” on page 5-3
- “C# Integration Examples” on page 5-28
- “Microsoft Visual Basic Integration Examples” on page 5-58
- “Component Access On Another Computer” on page 5-83
- “For More Information” on page 5-84

Common Integration Tasks

In “The Magic Square Component in an Enterprise C# Application” on page 1-21, and in “Integrate Your .NET Assembly Into a C# Application” on page 1-17 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 runtime 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.

Application Coding

In this section...

“Using C# Code In an Integrated .NET Component” on page 5-3

“Data Conversion” on page 5-4

“MATLAB API Functions in a C# Program” on page 5-16

“Object Passing by Reference” on page 5-17

“Real or Imaginary Components Within Complex Arrays” on page 5-20

“Jagged Array Processing” on page 5-22

“Field Additions to Data Structures and Data Structure Arrays” on page 5-22

“MATLAB Array Indexing” on page 5-22

“Block Console Display When Creating Figures” on page 5-23

“Error Handling” on page 5-25

“Explicitly Freeing Resources With Dispose” on page 5-26

Using C# Code In an Integrated .NET Component

Before you begin integrating your component code with your .NET application, it is helpful to understand how the elements of the library compiler 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 naming guidelines 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 MATLAB runtime.

Data Conversion

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.

See “Data Conversion Rules” on page 11-3 for complete tables detailing type-to-type data conversion rules using MATLAB Builder NE.

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 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: For information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

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” 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 11-3 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.

Multidimensional Array Processing in MATLAB and .NET

MATLAB and .NET implement different indexing strategies for multidimensional arrays. When you create a variable of type `MWNumericArray`, MATLAB automatically creates an equivalent array, using its own internal indexing. For example, MATLAB indexes using this schema:

```
(row column page1 page2 ...)
```

while .NET indexes as follows:

```
(... page2 page1 row column)
```

Given the multi-dimensional MATLAB `myarr`:

```
>> myarr(:,:,1) = [1, 2, 3; 4, 5, 6];
>> myarr(:,:,2) = [7, 8, 9; 10, 11, 12];
>> myarr
```

```
myarr(:,:,1) =
```

```
    1    2    3
    4    5    6
```

```
myarr(:,:,2) =
```

```
    7    8    9
   10   11   12
```

You would code this equivalent in .NET:

```
double[, ,] myarr = {{{1.000000, 2.000000, 3.000000},
{4.000000, 5.000000, 6.000000}}, {{7.000000, 8.000000,
```

```
9.000000}, {10.000000, 11.000000, 12.000000}}};
```

Manual Data Conversion from Native Types to MATLAB Types

- “Native Data Conversion” on page 5-8
- “Type Specification” on page 5-9
- “Optional Argument Specification” on page 5-9
- “Pass a Variable Number of Outputs” on page 5-11

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 11-9 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 5-5.

Type Specification

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
```

Optional Argument Specification

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.

Pass 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 11-3 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.

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

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

Return Value Handling

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 `MArray` class to query information about the underlying MATLAB array.

.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)
{
    MArray primes= null;
    MyPrimesClass myPrimesClass= null;
    try
    {
        myPrimesClass= new MyPrimesClass();
        primes= myPrimesClass.myprimes((double)n);
        Array primesArray= ((MNumericArray)primes).
            ToVector(MArrayComponent.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.

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.

MATLAB API Functions in a C# Program

- “Overview” on page 5-16
- “About Building Engine Applications ” on page 5-16
- “MATLAB Engine API in a C# Program” on page 5-16

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.

About Building Engine Applications

For detailed information about using an IDE to build engine applications, see *MATLAB External Interfaces*.

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("");
    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();
    }
}
}

```

Object Passing by Reference

- “MATLAB Array” on page 5-17
- “Wrapping and Passing .NET Objects with MWOBJECTARRAY” on page 5-18

MATLAB Array

MWOBJECTARRAY, a special subclass of MWARRAY, lets you create a MATLAB array that references .NET objects.

Note: For information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder`

\MWArrayAPI folder, where *matlabroot* represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

Wrapping and Passing .NET Objects with MWObjectArray

You can create a MATLAB code wrapper around .NET objects using `MWObjectArray`. Use this technique to pass objects by reference to MATLAB functions and return .NET objects. The examples in this section present some common use cases.

Passing a .NET Object into a MATLAB Builder NE Component

To pass an object into a MATLAB Builder NE component:

- 1 Write the MATLAB function that references a .NET type:

```
function addItem(hDictionary, key, value)
    if ~isa(hDictionary, 'System.Collections.Generic.IDictionary')
        error('foo:IncorrectType',
            ... 'expecting a System.Collections.Generic.Dictionary');
    end
    hDictionary.Add(key, value);
end
```

- 2 Create a .NET object to pass to the MATLAB function:

```
Dictionary char2Ascii= new Dictionary();
char2Ascii.Add("A", 65);
char2Ascii.Add("B", 66);
```

- 3 Create an instance of `MWObjectArray` to wrap the .NET object:

```
MWObjectArray MWchar2Ascii=
    new MWObjectArray(char2Ascii);
```

- 4 Pass the wrapped object to the MATLAB function:

```
myComp.addValue(MWchar2Ascii, 'C', 67);
```

Returning a Custom .NET Object in a MATLAB Function Using a Deployed .NET Builder Component

You can also use `MWObjectArray` to clone an object inside a MATLAB Builder NE component. Continuing with the example in “Passing a .NET Object into a MATLAB Builder NE Component” on page 5-18, perform the following steps:

- 1 Write the MATLAB function that references a .NET type:

```

function result= add(hMyDouble, value)

    if ~isa(hMyDouble, 'MyDoubleComp.MyDouble')
        error('foo:IncorrectType', 'expecting a MyDoubleComp.MyDouble');
    end
    hMyDoubleClone= hMyDouble.Clone();
    result= hMyDoubleClone.Add(value);

end

```

- 2 Create the object:

```
MyDouble myDouble= new MyDouble(75);
```

- 3 Create an instance of MWObjectArray to wrap the .NET object:

```
MWObjectArray MWdouble= new MWObjectArray(myDouble);
origRef = new MWObjectArray(hash);
```

- 4 Pass the wrapped object to the MATLAB function and retrieve the returned cloned object:

```
MWObjectArray result=
    (MWObjectArray)myComp.add(MWdouble, 25);
```

- 5 Unwrap the .NET object and print the result:

```
MyDouble doubleClone= (MyDouble)result.Object;

Console.WriteLine(myDouble.ToDouble());
Console.WriteLine(doubleClone.ToDouble());
```

Cloning an MWObjectArray

When calling the Clone method on MWObjectArray, the following rules apply for the wrapped object.

- If the wrapped object is a ValueType, it is deep-copied.
- If an object is not a ValueType and implements ICloneable, the Clone method for the object is called.
- The MemberwiseClone method is called on the wrapped object.

Calling Clone on MWObjectArray

```
MWObjectArray aDate = new MWObjectArray(new
    DateTime(1, 1, 2010));
MWObjectArray clonedDate = aDate.Clone();
```

Optimization Example Using `MWObjectArray`

For a full example of how to use `MWObjectArray` to create a reference to a .NET object and pass it to a component, see the “Optimization” (C#) and the “Optimization (Visual Basic)”.

`MWObjectArray` and Application Domains

Every ASP .NET Web application deployed to IIS is launched in a separate `AppDomain`.

The MATLAB .NET interface must support the .NET type wrapped by `MWObjectArray`. If the `MWObjectArray` is created in the default `AppDomain`, the wrapped type has no other restrictions.

If the `MWObjectArray` is not created in the default `AppDomain`, the wrapped .NET type must be serializable. This limitation is imposed by the fact that the object needs to be marshaled from the non-default `AppDomain` to the default `AppDomain` in order for MATLAB to access it.

Real or Imaginary Components Within Complex Arrays

- “Component Extraction” on page 5-20
- “Returning Values Using Component Indexing” on page 5-21
- “Assigning Values with Component Indexing” on page 5-21
- “Converting MATLAB Arrays to .NET Arrays Using Component Indexing” on page 5-21

Component Extraction

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 Using 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);
```

Jagged Array Processing

A *jagged array* is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes, as opposed to the elements of a *non-jagged array* whose elements are of the same dimensions and size.

Web services, in particular, process data almost exclusively in jagged arrays.

`MWNumericArrays` can only process jagged arrays with a rectangular shape.

In the following code snippet, a rectangular jagged array of type `int` is initialized and populated.

Initializing and Populating a Jagged Array

```
int[][] jagged = new int[5][];  
for (int i = 0; i < 5; i++)  
    jagged[i] = new int[10];  
MWNumericArray jaggedMWArray = new MWNumericArray(jagged);  
Console.WriteLine(jaggedMWArray);
```

Field Additions 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 “C# Integration Examples” on page 5-28 and “Microsoft Visual Basic Integration Examples” on page 5-58.

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 “Data Conversion” on page 16-8 for an example of native type conversion.

Block Console Display When Creating Figures

- “WaitForFiguresToDie Method” on page 5-23
- “Using WaitForFiguresToDie to Block Execution” on page 5-24

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.

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.

Error Handling

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;
    }
}
```

Explicitly Freeing Resources With Dispose

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`.

C# Integration Examples

In this section...
“Simple Plot” on page 5-28
“Passing Variable Arguments” on page 5-31
“Spectral Analysis” on page 5-36
“Matrix Math” on page 5-41
“Phone Book” on page 5-47
“Optimization” on page 5-52

Simple Plot

- “Purpose” on page 5-28
- “Procedure” on page 5-28

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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- 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\VSVersion\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 From the MATLAB apps gallery, open the **Library Compiler** app.
- 4 Build the .NET component. See the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11 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\VSVersion\PlotExample\PlotCSApp\PlotApp.cs`.

The program listing is shown here.

```
// *****
//
// PlotApp.cs
//
// This example demonstrates how to use MATLAB Builder NE to build a component
// that displays a MATLAB figure window.
//
// Copyright 2001-2012 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 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” 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” on page 5-28, 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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

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

Procedure 5.3. 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\VSVersion\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 2001-2012 The MathWorks, Inc.

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

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

- 3 From the MATLAB apps gallery, open the **Library Compiler** app.
- 4 Build the .NET component. See the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11 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 2001-2012 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 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” 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

- “Purpose” on page 5-36
- “Procedure” on page 5-37

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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where *matlabroot* represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- 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\VSVersion\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 2001-2012 The MathWorks, Inc.
if (isempty(data))
    fftdata = [];
    freq = [];
    powerspect = [];
    return;
end
if (interval <= 0)
    error('Sampling interval must be greater than zero');
    return;
end
fftData = fft(data);
freq = (0:length(fftData)-1)/(length(fftData)*interval);
powerSpect = abs(fftData)/(sqrt(length(fftData)));

```

plotfft.m

```

function 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 2001-2012 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\VSVersion\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 From the MATLAB apps gallery, open the **Library Compiler** app.
- 4 Build the .NET component. See the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11 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 2001-2012 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  $\sin(2\pi \cdot 15 \cdot t)$  +  $\sin(2\pi \cdot 40 \cdot t)$  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 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” 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\bin\distrib` subfolder of your work area.)
- 7 Build and run the application in Visual Studio .NET.

Matrix Math

- “Purpose” on page 5-41
- “Procedure” on page 5-42
- “MATLAB Functions to Be Encapsulated” on page 5-46
- “Understanding the MatrixMath Program” on page 5-47

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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your

MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- 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:

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

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\VSVersion\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`, `ludcomp`, and `qrdecomp` functions is already in your work folder in `MatrixMathExample\MatrixMathComp\`.

- 3 From the MATLAB apps gallery, open the **Library Compiler** app.
- 4 Build the .NET component. See the instructions in "Create a .NET Assembly From MATLAB Code" on page 1-11 for more details. Use the following information:

Project Name	<code>MatrixMathComp</code>
Class Name	<code>Factor</code>
Files to compile	<code>cholesky ludcomp qrdecomp</code>

- 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.cs
// 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 2001-2012 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 5-47 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 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`

\marray.dll. See “Supported Microsoft .NET Framework Versions” 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 2001-2012 The MathWorks, Inc.

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 2001-2012 The MathWorks, Inc.

[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 2001-2012 The MathWorks, Inc.
```



```
[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.

Phone Book

- “Purpose” on page 5-47
- “Procedure” on page 5-48

Purpose

The `makephone` function takes a structure array as an input, modifies it, and supplies the modified array as an output.

Note: For information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

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\toolbox\dotnetbuilder\Examples\VSVersion\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 2006-2012 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 From the MATLAB apps gallery, open the **Library Compiler** app.
- 4 Build the .NET component. See the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11 for more details. Use the following information:

Project Name	PhoneBookComp
Class Name	Phonebook
File to compile	makephone

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

The sample application for this example is in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\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 2001-2012 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; /* Ouput 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");
            }
        }
    }
}
```



```
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
```

Optimization

- “Purpose” on page 5-53
- “OptimizeComp Component” on page 5-53
- “Procedure” on page 5-53

Purpose

This example shows how to:

- Use the MATLAB Builder NE product to create a component (`OptimizeComp`). This component applies MATLAB optimization routines to objective functions implemented as .NET objects.
- Access the component in a .NET application (`OptimizeApp.cs`). Then use the `MWObjectArray` class to create a reference to a .NET object (`BananaFunction.cs`), and pass that object to the component.

Note: For information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- Build and run the application.

OptimizeComp Component

The component (`OptimizeComp`) finds a local minimum of an objective function and returns the minimal location and value. The component uses the MATLAB optimization function `fminsearch`. This example optimizes the Rosenbrock banana function used in the `fminsearch` documentation.

The class `OptimizeComp.OptimizeClass` performs an unconstrained nonlinear optimization on an objective function implemented as a .NET object. A method of this class, `doOptim`, accepts an initial value (NET object) that implements the objective function, and returns the location and value of a local minimum.

The second method, `displayObj`, is a debugging tool that lists the characteristics of a .NET object. These two methods, `doOptim` and `displayObj`, encapsulate MATLAB functions. The MATLAB code for these two methods resides in `doOptim.m` and `displayObj.m`. You can find this code in `matlabroot\toolbox\dotnetbuilder\VSVersion\NET\Examples\OptimizeExample\OptimizeComp`.

Procedure

- 1 If you have not already done so, copy the files for this example as follows:
 - 1 Copy the following folder that ships with MATLAB to your work folder:

```
matlabroot\toolbox\dotnetbuilder\VSVersion\NET\Examples
\OptimizeExample
```

- 2 At the MATLAB command prompt, `cd` to the new `OptimizeExample` subfolder in your work folder.
- 2 If you have not already done so, set the environment variables that are required on a development machine. See “MATLAB Runtime Path Settings for Development and Testing” on page 6-5.
- 3 Write the MATLAB code that you want to access. This example uses `doOptim.m` and `displayObj.m`, which already reside in your work folder. The path is `matlabroot\toolbox\dotnetbuilder\VSVersion\NET\Examples\OptimizeExample\OptimizeComp`.

For reference, the code of `doOptim.m` is displayed here:

```
function [x,fval] = doOptim(h, x0)
mWrapper = @(x) h.evaluateFunction(x);
```

```
directEval = h.evaluateFunction(x0)
wrapperEval = mWrapper(x0)
```

```
[x,fval] = fminsearch(mWrapper,x0)
```

For reference, the code of `displayObj.m` is displayed here:

```
function className = displayObj(h)
```

```
h
className = class(h)
whos('h')
methods(h)
```

- 4 From the MATLAB apps gallery, open the **Library Compiler** app.
- 5 As you compile the .NET application using the Library Compiler, use the following information:

Project Name	OptimizeComp
Class Name	OptimizeComp.OptimizeClass
File to compile	doOptim.m displayObj.m

- 6 Write source code for a class (`BananaFunction`) that implements an object function to optimize. The sample application for this example is in `matlabroot\toolbox\dotnetbuilder\VSVersion\NET\Examples\OptimizeExample`

\OptimizeCSApp. The program listing for `BananaFunction.cs` displays the following code:

```
// *****//
//BananaFunction.cs
//
// This file is used as an example for the MATLAB Builder NE product.
//
// It implements the Rosenbrock banana function described in the FMINSEARCH
// documentation
//
// Copyright 2001-2010 The MathWorks, Inc.
//
// *****//
using System;

namespace MathWorks.Examples.Optimize
{
    public class BananaFunction
    {
        public BananaFunction() {}

        public double evaluateFunction(double[] x)
        {
            double term1= 100*Math.Pow((x[1]-Math.Pow(x[0],2.0)),2.0);
            double term2= Math.Pow((1-x[0]),2.0);
            return term1+term2;
        }
    }
}
```

The class implements the Rosenbrock banana function described in the `fminsearch` documentation.

- 7 Customize the application using Visual Studio .NET using the `OptimizeCSApp` folder, which contains a Visual Studio .NET project file for this example.
 - a . Open the project in Visual Studio .NET by double-clicking `OptimizeCSApp.csproj` in Windows Explorer. You can also open it from the desktop by right-clicking **OptimizeCSApp.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 `OptimizeComp` component which you built in a previous step. (The component, `OptimizeComp.dll`, is in the `\OptimizeExample\OptimizeComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

When run successfully, the program displays the following output:

```
Using initial points= -1.2000 1
```

```

*****
**          Properties of .NET Object          **
*****

h =

    MathWorks.Examples.Optimize.BananaFunction handle
        with no properties.
    Package: MathWorks.Examples.Optimize

className =

MathWorks.Examples.Optimize.BananaFunction

    Name  Size  Bytes  Class  Attributes

    h      1x1   60  MathWorks.Examples.Optimize.BananaFunction

Methods for class MathWorks.Examples.Optimize.BananaFunction:

BananaFunction  addlistener  findprop  lt
Equals          delete       ge        ne
GetHashCode     eq          gt        notify
GetType         evaluateFunction  isvalid
ToString       findobj     le

***** Finished displayObj *****

*****
** Performing unconstrained nonlinear optimization **
*****

directEval =

```

24.2000

wrapperEval =

24.2000

x =

1.0000 1.0000

fval =

8.1777e-010

***** Finished doOptim *****

Location of minimum: 1.0000 1.0000
Function value at minimum: 8.1777e-010

Microsoft Visual Basic Integration Examples

In this section...
“Magic Square (Visual Basic)” on page 5-58
“Create Plot Example (Visual Basic)” on page 5-61
“Variable Arguments (Visual Basic)” on page 5-63
“Spectral Analysis (Visual Basic)” on page 5-66
“Matrix Math (Visual Basic)” on page 5-69
“Phone Book (Visual Basic)” on page 5-73
“Optimization (Visual Basic)” on page 5-78

Note: The examples for the MATLAB Builder NE product are in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion`, where `matlabroot` is the folder where the MATLAB product is installed and `VSVersion` specifies the version of Microsoft Visual Studio .NET you are using. 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\VSVersion\DotNetExamples.sln
```

Note: The sample applications that follow use the same components as those developed in “Create a .NET Assembly From MATLAB Code” on page 1-11 and “C# Integration Examples” on page 5-28. 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 (Visual Basic)

To create the component for this example, see the first several steps in “Create a .NET Assembly From MATLAB Code” on page 1-11. 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\VSVersion\NET\MagicSquareExample\MagicSquareVBAApp

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 2001-2012 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
            End Try
        End Sub
    End Class
End Namespace
```

```
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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click [this link MWArray Class Library Reference](#).

- 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 MagicSquareVApp folder contains a Visual Studio .NET project file for each example. Open the project in Visual Studio .NET for this example by double-clicking MagicSquareVApp.vbproj in Windows Explorer.
 - b Add a reference to the MWArray component, which is `matlabroot\toolbox\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” on page 5-28. Then create a Visual Basic application as follows:

- 1 Review the sample application for this example in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PlotExample\PlotVApp\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 2001-2012 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 information about these data conversion classes, see the *MATLAB MArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MArray Class Library Reference](#).

- 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 `PlotVApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PlotVApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **PlotVApp.vbproj** > **Open Outside MATLAB**.
 - b Add a reference to the `MArray` component, which is `matlabroot\toolbox\dotnetbuilder\bin\architecture\framework_version\marray.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 (Visual Basic)

To create the component for this example, see “Passing Variable Arguments” on page 5-31. Then create a Microsoft Visual Basic application as follows:

- 1 Review the sample application for this example in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\VarArgExample\VarArgVApp\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 2001-2012 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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- 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 VarArgVApp folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking VarArgVApp.vbproj in Windows Explorer. You can also open it from the desktop by right-clicking **VarArgVApp.vbproj** > **Open Outside MATLAB**.
 - b Add a reference to the MArray component, which is `matlabroot\toolbox\dotnetbuilder\bin\architecture\framework_version\marray.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 (Visual Basic)

To create the component for this example, see the first few steps of the “Spectral Analysis” on page 5-36. Then create a Microsoft Visual Basic application as follows:

- 1 Review the sample application for this example in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\SpectraExample\SpectraVApp\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 2001-2012 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
            End Try
        End Sub
    End Class
End Namespace

```

```
        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 information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- 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 SpectraVBAApp folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `SpectraVBAApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **SpectraVBAApp.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 (Visual Basic)

To create the component for this example, see the first few steps in “Matrix Math” on page 5-41. Then create a Microsoft Visual Basic application as follows.

- 1 Review the sample application for this example in:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MatrixMathExample\MatrixMathVBAApp\MatrixMathApp
```

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 2001-2012 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, _
    MNNumericArray.MakeSparse(matrixOrder, matrixOrder,
        MWArrayComplexity.Real,
        (matrixOrder + 2 * (matrixOrder - 1))), _
    New MNNumericArray(MWArrayComplexity.Real, MNNumericType.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 MNNumericArray(2.0)
        ElseIf colIdx = rowIdx + 1 Or colIdx = rowIdx - 1 Then
            matrix(rowIdx, colIdx) = New MNNumericArray(-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
argsOut = factor.cholesky(matrix)

Console.WriteLine("Cholesky Factorization:{0}{1}{2}",
    Chr(10), argsOut, 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))

MNNumericArray.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 5-47 for more details about the structure of this program.

- 2 Build the `MatrixMathApp` application using Visual Studio .NET.
 - a The `MatrixMathVBAApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `MatrixMathVBAApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **MatrixMathVBAApp.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.

Phone Book (Visual Basic)

- “makephone Function” on page 5-73
- “Procedure” on page 5-73

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` API documentation.

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\toolbox\dotnetbuilder\Examples\VSVersion\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 2006-2012 The MathWorks, Inc.

book = friends;
for i = 1:numel(friends)
    numberStr = num2str(book(i).phone);
    book(i).external = ['(508) 555-' numberStr];
end
```

end

This code is already in your work folder in PhoneBookExample\PhoneBookComp\makephone.m.

- 3 From the MATLAB apps gallery, open the **Library Compiler** app.
- 4 Build the .NET component. See the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11 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\VSVersion\NET* PhoneBookExample\PhoneBookVApp\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 2001-2012 The MathWorks, Inc.
' *****

' Necessary package imports

Imports MathWorks.MATLAB.NET.Arrays
Imports PhoneBookComp

'
' getphone class demonstrates the use of the MWStructArray class
'
Public Module PhoneBookVApp
    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      'Ouput 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.WriteLine("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 MArray = 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.WriteLine("    " + 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 the `Phonebook` 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 desktop by right-clicking **PhoneBookVBApp.vbproj** > **Open Outside MATLAB**.
 - b Add a reference to the `MArray` component, which is `matlabroot\toolbox\dotnetbuilder\bin\architecture\framework_version\marray.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
```

Optimization (Visual Basic)

Optimization Example

- “Purpose” on page 5-78
- “OptimizeComp Component” on page 5-78
- “Procedure” on page 5-79

Purpose

This example shows how to:

- Use the MATLAB Builder NE product to create a component (`OptimizeComp`). This component applies MATLAB optimization routines to objective functions implemented as .NET objects.
- Access the component in a .NET application (`OptimizeApp.vb`). Then, use the `MWObjectArray` class to create a reference to a .NET object (`BananaFunction.vb`), and pass that object to the component.

Note: For information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

- Build and run the application.

OptimizeComp Component

The component (`OptimizeComp`) finds a local minimum of an objective function and returns the minimal location and value. The component uses the MATLAB optimization function `fminsearch`. This example optimizes the Rosenbrock banana function used in the `fminsearch` documentation.

The class `OptimizeComp.OptimizeClass` performs an unconstrained nonlinear optimization on an objective function implemented as a .NET object. A method of this class, `doOptim`, accepts an initial value (NET object) that implements the objective function, and returns the location and value of a local minimum.

The second method, `displayObj`, is a debugging tool that lists the characteristics of a .NET object. These two methods, `doOptim` and `displayObj`, encapsulate MATLAB functions. The MATLAB code for these two methods resides in `doOptim.m` and

`displayObj.m`. You can find this code in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample\OptimizeVBAApp`.

Procedure

- 1 If you have not already done so, copy the files for this example as follows:
 - 1 Copy the following folder that ships with MATLAB to your work folder: `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample`
 - 2 At the MATLAB command prompt, `cd` to the new `OptimizeExample` subfolder in your work folder.
- 2 If you have not already done so, set the environment variables that are required on a development machine. See “MATLAB Runtime Path Settings for Development and Testing” on page 6-5.
- 3 Write the MATLAB code that you want to access. This example uses `doOptim.m` and `displayObj.m`, which already resides in your work folder. The path is `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample\OptimizeComp`.

For reference, the code of `doOptim.m` is displayed here:

```
function [x,fval] = doOptim(h, x0)
mWrapper = @(x) h.evaluateFunction(x);
```

```
directEval = h.evaluateFunction(x0)
wrapperEval = mWrapper(x0)
```

```
[x,fval] = fminsearch(mWrapper,x0)
```

For reference, the code of `displayObj.m` is displayed here:

```
function className = displayObj(h)
```

```
h
className = class(h)
whos('h')
methods(h)
```

- 4 From the MATLAB apps gallery, open the **Library Compiler** app.
- 5 As you compile the .NET application using the Library Compiler, use the following information:

Project Name

OptimizeComp

Class Name	OptimizeComp.OptimizeClass
File to compile	doOptim.m displayObj.m

- 6 Write source code for a class (`BananaFunction`) that implements an object function to optimize. The sample application for this example is in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample\OptimizeVBAApp`. The program listing for `BananaFunction.vb` displays the following code:

```
' *****  
'  
' BananaFunction.vb  
'  
' This file is used as an example for the MATLAB Builder NE product.  
'  
' It implements the Rosenbrock banana function described in the FMINSEARCH  
' documentation  
'  
' Copyright 2001-2009 The MathWorks, Inc.  
' *****  
  
Imports System  
  
Namespace MathWorks.Examples.Optimize  
  
    Class BananaFunction  
  
#Region "Methods"  
        Public Sub BananaFunction()  
            End Sub  
  
        Public Function evaluateFunction(ByVal x As Double()) As Double  
  
            Dim term1 As Double = 100 * Math.Pow((x(1) - Math.Pow(x(0),  
                2.0)), 2.0)  
  
            Dim term2 As Double = Math.Pow((1 - x(0)), 2.0)  
            Return term1 + term2  
  
        End Function  
#End Region  
  
    End Class  
End Namespace
```

The class implements the Rosenbrock banana function described in the `fminsearch` documentation.

- 7 Customize the application using Visual Studio .NET using the `OptimizeVBAApp` folder, which contains a Visual Studio .NET project file for this example.

- a The OptimizeVbApp folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `OptimizeVbApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **OptimizeVbApp.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 `OptimizeComp` component which you built in a previous step. (The component, `OptimizeComp.dll`, is in the `\OptimizeExample\OptimizeComp\x86\V2.0\Debug\distrib` subfolder of your work area.)

When run successfully, the program displays the following output:

```
Using initial points= -1.2000 1
```

```
*****
**                Properties of .NET Object                **
*****
```

```
h =
```

```
MathWorks.Examples.Optimize.BananaFunction handle w
ith no properties.
Package: MathWorks.Examples.Optimize
```

```
className =
```

```
MathWorks.Examples.Optimize.BananaFunction
```

Name	Size	Bytes	Class	Attributes
h	1x1	60	MathWorks.Examples.Optimize.BananaFunction	

```
Methods for class MathWorks.Examples.Optimize.BananaFunction:
```

```
BananaFunction  addlistener  findprop  lt
Equals          delete       ge        ne
GetHashCode     eq         gt        notify
GetType        evaluateFunction  isvalid
ToString       findobj   le
```

```
***** Finished displayObj *****
```

```
*****
** Performing unconstrained nonlinear optimization **
*****
```

```
directEval =
```

```
24.2000
```

```
wrapperEval =
```

```
24.2000
```

```
x =
```

```
1.0000  1.0000
```

```
fval =
```

```
8.1777e-010
```

```
***** Finished doOptim *****
```

```
Location of minimum: 1.0000  1.0000
Function value at minimum: 8.1777e-010
```

Component Access 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 “Create a .NET Assembly From MATLAB Code” on page 1-11.

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” 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.

Note: For information about these data conversion classes, see the *MATLAB MWArray Class Library Reference*, available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder. If you set your help preference to read locally installed documentation, click this link [MWArray Class Library Reference](#).

To avoid using data conversion classes, see “Implementing a Type-Safe Interface” on page 7-10.

For More Information

If you want to...	See...
Learn how to build a component	“Create a .NET Assembly From MATLAB Code” on page 1-11
Perform basic integration tasks using C# code	“Integrate Your .NET Assembly Into a C# Application” on page 1-17
<ul style="list-style-type: none">• Basic MATLAB Programmer tasks• How the deployment products process your MATLAB functions• How the deployment products work together	“Write Deployable MATLAB Code” on page 2-10
Learn about supported MATLAB Builder NE targets	“Supported Compilation Targets” on page 3-2
Work with cell arrays and data structures using native .NET types	“Using Native .NET Structure and Cell Arrays” on page 9-7

Distribute to End Users

- “Deploying Components to End Users” on page 6-2
- “MATLAB Runtime Path Settings for Development and Testing” on page 6-5
- “MATLAB Runtime Path Settings for Run-time Deployment” on page 6-7
- “MATLAB Runtime Run-Time Options” on page 6-9
- “MATLAB Runtime Component Cache and Deployable Archive Embedding” on page 6-11
- “The MATLAB Runtime User Data Interface” on page 6-13
- “Impersonation Implementation Using ASP.NET” on page 6-18
- “Enhanced XML Documentation Files” on page 6-22

Deploying Components to End Users

Distributing MATLAB Code Using the MATLAB Runtime

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

Install MATLAB Runtime

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

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

The MATLAB runtime installer does the following:

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

MATLAB Runtime Prerequisites

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

Add the MATLAB Runtime Installer to the Installer

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

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

Install the MATLAB Runtime

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

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

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

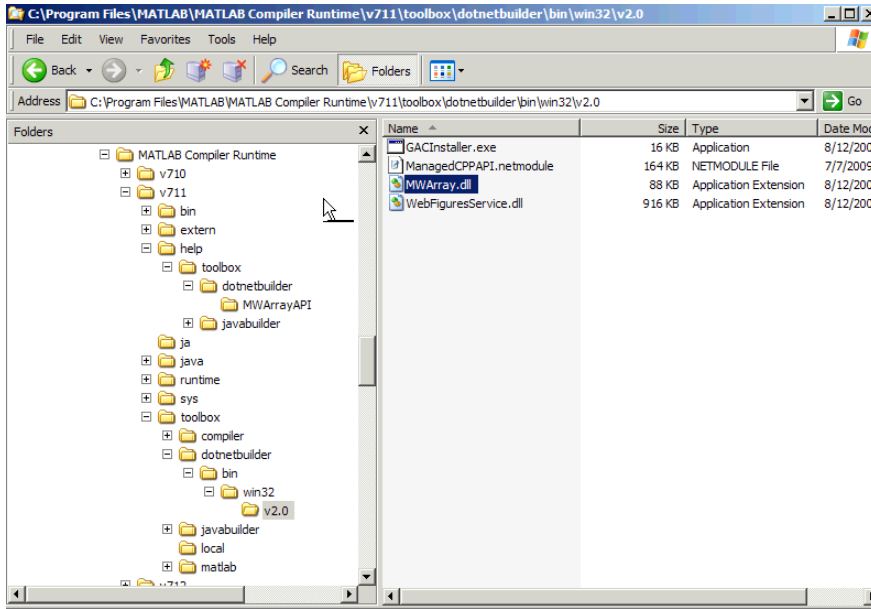
Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See “Using MATLAB Compiler on Mac or Linux” for detailed information on performing all deployment tasks specifically with UNIX variants such as Linux and Mac.

Where to find the MWArray API

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

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

See the *MATLAB Builder NE Release Notes* for a list of supported framework versions.



Sample Directory Structure of the MATLAB Runtime Including MWArray.dll

MATLAB Runtime Path Settings for Development and Testing

In this section...

“Overview” on page 6-5

“Path for Java Development on All Platforms” on page 6-5

“Path Modifications Required for Accessibility” on page 6-5

“Windows Settings for Development and Testing” on page 6-6

“Linux Settings for Development and Testing” on page 6-6

“Mac Settings for Development and Testing” on page 6-6

Overview

The following information is for developers developing applications that use compiled MATLAB code. These settings are required on the machine where you are developing your application.

Note: For *matlabroot*, substitute the MATLAB root folder on your system. Type *matlabroot* to see this folder name.

Path for Java Development on All Platforms

There are additional requirements when programming in the Java programming language.

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.

Windows Settings for Development and Testing

When programming with compiled MATLAB code, add the following folder to your system PATH environment variable:

```
matlabroot\runtime\win32|win64
```

Linux Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

```
setenv LD_LIBRARY_PATH
matlabroot/runtime/glnxa64:
matlabroot/bin/glnxa64:
matlabroot/sys/os/glnxa64:
mcr_root/version/sys/opencv/lib/glnxa64
```

Mac Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

```
setenv DYLD_LIBRARY_PATH
matlabroot/runtime/maci64:
matlabroot/bin/maci64:
matlabroot/sys/os/maci64:
```

MATLAB Runtime Path Settings for Run-time Deployment

In this section...

“General Path Guidelines” on page 6-7

“Path for Java Applications on All Platforms” on page 6-7

“Windows Path for Run-Time Deployment” on page 6-7

“Linux Paths for Run-Time Deployment” on page 6-8

“Mac Paths for Run-Time Deployment” on page 6-8

General Path Guidelines

Regardless of platform, be aware of the following guidelines with regards to placing specific folders on the path:

- Always avoid including `bin` or `arch` on the path. Failure to do so may inhibit ability to run multiple MATLAB runtime instances.
- Ideally, set the environment in a separate shell script to avoid run-time errors caused by path-related issues.

Path for Java Applications on All Platforms

When your users run applications that contain compiled MATLAB code, you must instruct them to set the path so that the system can find the MATLAB runtime.

Note: When you deploy a Java application to end users, they must set the class path on the target machine.

The system needs to find `.jar` files containing the MATLAB libraries. To tell the system how to locate the `.jar` files it needs, specify a `classpath` either in the `javac` command or in your system environment variables.

Windows Path for Run-Time Deployment

The following folder should be added to the system path:

```
mcr_root\version\runtime\win32|win64
```

mcr_root refers to the complete path where the MATLAB runtime library archive files are installed on the machine where the application is to be run.

mcr_root is version specific; you must determine the path after you install the MATLAB runtime.

Note: If you are running the MATLAB Runtime Installer on a shared folder, be aware that other users of the share may need to alter their system configuration.

Linux Paths for Run-Time Deployment

Use these `setenv` commands to set your MATLAB runtime run-time paths.

```
setenv LD_LIBRARY_PATH
  mcr_root/version/runtime/glnxa64:
  mcr_root/version/bin/glnxa64:
  mcr_root/version/sys/os/glnxa64:
  mcr_root/version/sys/opengl/lib/glnxa64
```

Mac Paths for Run-Time Deployment

Use these `setenv` commands to set your MATLAB runtime run-time paths.

```
setenv DYLD_LIBRARY_PATH
  mcr_root/version/runtime/maci64:
  mcr_root/version/bin/maci64:
  mcr_root/version/sys/os/maci64:
  mcr_root/version/sys/java/jar/maci64/jre/lib/server
```

MATLAB Runtime Run-Time Options

In this section...

“What Run-Time Options Can You Specify?” on page 6-9

“Getting MATLAB Runtime Option Values Using MWMCR” on page 6-9

What Run-Time Options Can You Specify?

As of R2009a, you can pass MATLAB runtime 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 MATLAB runtime attributes and MATLAB runtime state.

Getting MATLAB Runtime Option Values Using MWMCR

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

MWMCR Method	Purpose
<code>MWMCR.IsMCRInitialized()</code>	Returns <code>true</code> if the runtime is initialized, otherwise returns <code>false</code> .
<code>MWMCR.IsMCRJVMEabled()</code>	Returns <code>true</code> if the runtime is launched with .NET Virtual Machine (JVM), otherwise returns <code>false</code> .
<code>MWMCR.GetMCRLogFileName()</code>	Returns the name of the log file passed with the <code>LOGFILE</code> attribute.

Default MATLAB Runtime Options

If you pass no options, the MATLAB runtime is launched with default option values:

MATLAB Runtime Run-Time Option	Default Option Values
.NET Virtual Machine (JVM)	<code>NOJVM(false)</code>
Log file usage	<code>LOGFILE(null)</code>

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

Use the following attributes to represent the MATLAB runtime options you want to modify.

MWMCR Attribute	Purpose
NOJVM	Lets users launch the runtime with or without a JVM. It takes a Boolean as input. For example, <code>NOJVM(true)</code> launches the runtime without a JVM.
LOGFILE	Lets users pass the name of a log file, taking the file name as input. For example, <code>LOGFILE("logfile3.txt")</code> .

Passing MATLAB Runtime Option Values from a C# Application

Following is an example of how MATLAB runtime 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);
            }
        }
    }
}
```


MATLAB Runtime Component Cache and Deployable Archive Embedding

In this section...

“Overriding Default Behavior” on page 6-12

“For More Information” on page 6-12

Deployable archive data is automatically embedded directly in .NET and COM components and extracted to a temporary folder.

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

These variables allow you to specify the following:

- Define the default location where you want the deployable archive to be automatically extracted
- Add diagnostic error printing options that can be used when automatically extracting the deployable archive, for troubleshooting purposes
- Tuning the MATLAB runtime component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location.	Does not apply
MCR_CACHE_VERBOSE	When set to any value, this variable prints logging details about the component cache for diagnostic reasons. This can be very helpful if problems are encountered during deployable archive extraction.	Logging details are turned off by default (for example, when this variable has no value).

Environment Variable	Purpose	Notes
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mcrcachedir</code> command, with the desired cache size limit.

You can override this automatic embedding and extraction behavior by compiling with the “Overriding Default Behavior” on page 6-12 option.

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

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

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

Overriding Default Behavior

To extract the deployable archive in a manner prior to R2008b, alongside the compiled `.NET` or `COM` component, compile using the option `mcc`'s `-C` option.

You might want to use this option to troubleshoot problems with the deployable archive, for example, as the log and diagnostic messages are much more visible.

For More Information

For more information about the deployable archive, see “Deployable Archive”.

The MATLAB Runtime User Data Interface

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

- You need to supply run-time profile information to a client running an application created with the Parallel Computing Toolbox™ software. Profiles may be supplied (and changed) on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles.
- You want to initialize the MATLAB runtime 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-MATLAB runtime instance state access through an object-oriented API. Unlike MATLAB Compiler, access to a per-runtime 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 “Supplying Cluster Profiles for Parallel Computing Toolbox Applications” on page 6-13.

For more information, see “Using the MATLAB Runtime User Data Interface” in the MATLAB Compiler User's Guide.

Supplying Cluster Profiles for Parallel Computing Toolbox Applications

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

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

Step 1: Write Your Parallel Computing Toolbox Code

- 1 Compile `sample_pct.m` in MATLAB.

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

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

```
function speedup = sample_pct (n)
warning off all;
tic
if(ischar(n))
    n=str2double(n);
end
for ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time1 =toc;
parpool;
tic
parfor ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time2 =toc;
disp(['Normal loop times: ' num2str(time1) ...
    ',parallel loop time: ' num2str(time2) ]);
disp(['parallel speedup: ' num2str(1/(time2/time1)) ...
    ' times faster than normal']);
delete(gcf);
disp('done');
speedup = (time1/time2);
```

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

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

- 3 Verify that you get the following results;

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

Step 2: Set the Parallel Computing Toolbox Profile

In order to compile MATLAB code to a .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 profile once. You then call your other functions to utilize the Parallel Computing Toolbox functions.

Create the following `init` function:

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

Step 3: Compile Your Function

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 Library Compiler as follows:

- 1 Follow the steps in “Create a .NET Assembly From MATLAB Code” on page 1-11 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

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

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

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

- 2 To deploy the compiled application, copy the `for_redistribution_files_only` folder, which contains the following, to your end users.
 - `netPctComp.dll`
 - `MWArray.dll`
 - MATLAB Runtime Installer
 - Cluster profile

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 “Integrate Your .NET Assembly Into a C# Application” on page 1-17 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:



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

Impersonation Implementation Using ASP.NET

When running third-party software (for example, SQL Server[®]) there are times when it is necessary to use *impersonation* to perform Windows authentication in an ASP.NET application.

In deployed applications, impersonated credentials are passed in from IIS. However, since impersonation operates on a per-thread basis, this can sometimes present problems when processing the MATLAB runtime thread in a multi-threaded deployed application.

Use the following examples to turn impersonation on and off in your MATLAB file, to avoid problems stemming from MATLAB runtime thread processing issues.

Turning On Impersonation in a MATLAB MEX-file

```
#include mex.h
#include windows.h

/*
 *This mex function is called with a single int which
 *represents the user
 *identity token. We use this token to impersonate a
 *user on the interpreter
 *thread. This acts as a workaround for ASP.NET
 *applications that use
 *impersonation to pass the proper credentials
 *to SQL Server for windows
 *authentication. The function returns non zero status
 *for success, zero otherwise.
 */
void mexFunction( int          nlhs,
                  mxArray *    plhs[],
                  int          nrhs,
                  const mxArray * prhs[] )
{
    plhs[0] = mxCreateDoubleScalar(0); //return status

    HANDLE hToken =
        reinterpret_cast(*(mwSize *)mxGetData(prhs[0]));

    if(nrhs != 1)
    {
```



```

        mexErrMsgTxt("Incorrect number of input argument(s).
                    Expecting 1.");
    }

    int hr;

    if(!(hr = ImpersonateLoggedOnUser(hToken)))
    {
        mexErrMsgTxt("Error impersonating.\n");
    }

    *(mxGetPr(plhs[0])) = hr;
}

```

Turning Off Impersonation in a MATLAB MEX-file

```

#include mex.h
#include windows.h

/*
 *This mex function reverts to the old identity on the
 interpreter thread */
void mexFunction( int          nlhs,
                  mxArray *    plhs[],
                  int          nrhs,
                  const mxArray * prhs[] )
{
    if(!RevertToSelf())
    {
        mexErrMsgTxt("Failed to revert to the old
                    identity.");
    }
}

```

Code Added to Support Impersonation in ASP.NET Application

```

Monitor.Enter(someObj);

DeployedComponent.DeployedComponentClass myComp;

```

```
try
{
    System.Security.Principal.WindowsIdentity myIdentity =
        System.Security.Principal.WindowsIdentity.GetCurrent();

    //short circuit if user app is not impersonated
    if(myIdentity.isImpersonated())
    {
        myComp = new DeployedComponent.
            DeployedComponentClass ();

        //Run Users code

        MWArray[] output = myComp.impersonateUser(1,
            getToken());
    }
    else
    {
        //Run Users code
    }
}
Catch(Exception e)
{
}
finally
{
    if(myComp!=null)
        myComp.stopImpersonation();
    Monitor.Exit(someObj);
}

//
//
//Utility method to read the token for the current user
//and wraps it in a MWArray private MWNumericArray getToken()
{
    System.Security.Principal.WindowsIdentity myIdentity =
        System.Security.Principal.WindowsIdentity.GetCurrent();

    MWNumericArray a = null;

    if (IntPtr.Size == 4)
    {
```

```
        int intToken = myIdentity.Token.ToInt32();
        a = new MWNumericArray(intToken, false);
    }
    else
    {
        Int64 intToken = myIdentity.Token.ToInt64();
        a = new MWNumericArray(intToken, false);
    }
    return a;
```

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.

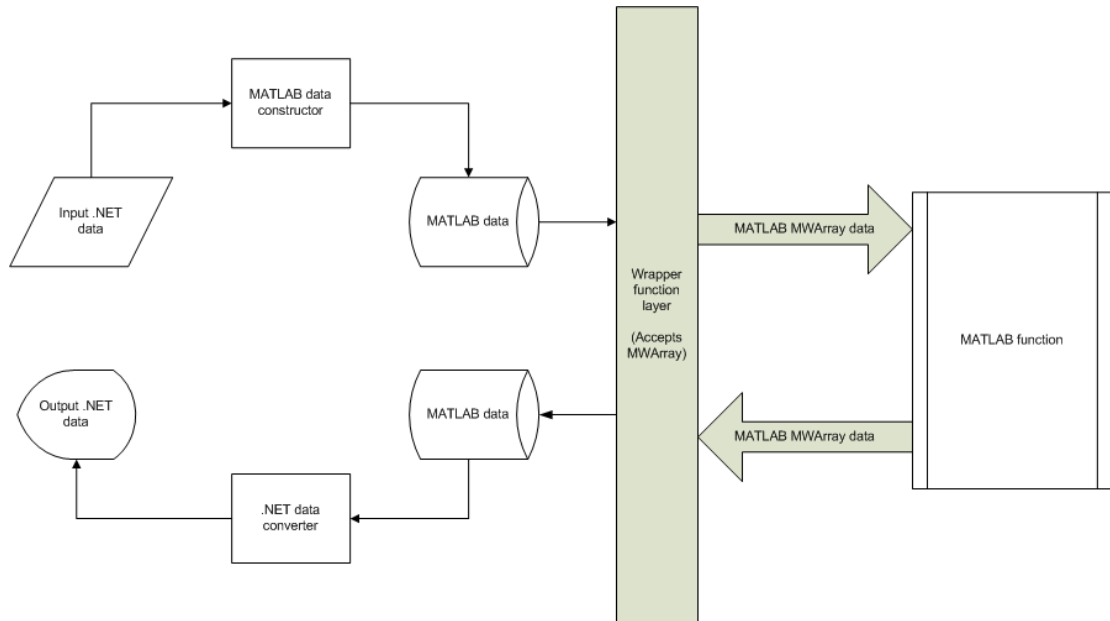
Type-Safe Interfaces, WCF, and MEF

- “Type-Safe Interfaces: An Alternative to Manual Data Marshaling” on page 7-2
- “Advantages of Implementing a Type-Safe Interface” on page 7-4
- “How Type-Safe Interfaces Work” on page 7-5
- “Build Your Component and Generate Your Type-Safe API” on page 7-8
- “Implementing a Type-Safe Interface” on page 7-10
- “Create Windows Communications Foundation (WCF)TM-Based Components” on page 7-12
- “Create Managed Extensibility Framework (MEF) Plug-Ins” on page 7-25

Type-Safe Interfaces: An Alternative to Manual Data Marshaling

The MATLAB data types are incompatible with native .NET types. To send data between your application and .NET, you perform these tasks:

- 1 Marshal data from .NET input data to a deployed function by creating an `MWArray` object from native .NET data. The public functions in a deployed component return `MWArray` objects.
- 2 Marshal the output MATLAB data in an `MWArray` into native .NET data by calling one of the `MWArray` marshaling methods (`ToArray()`, for example).



Manual Data Marshaling Without a Type-Safe Interface

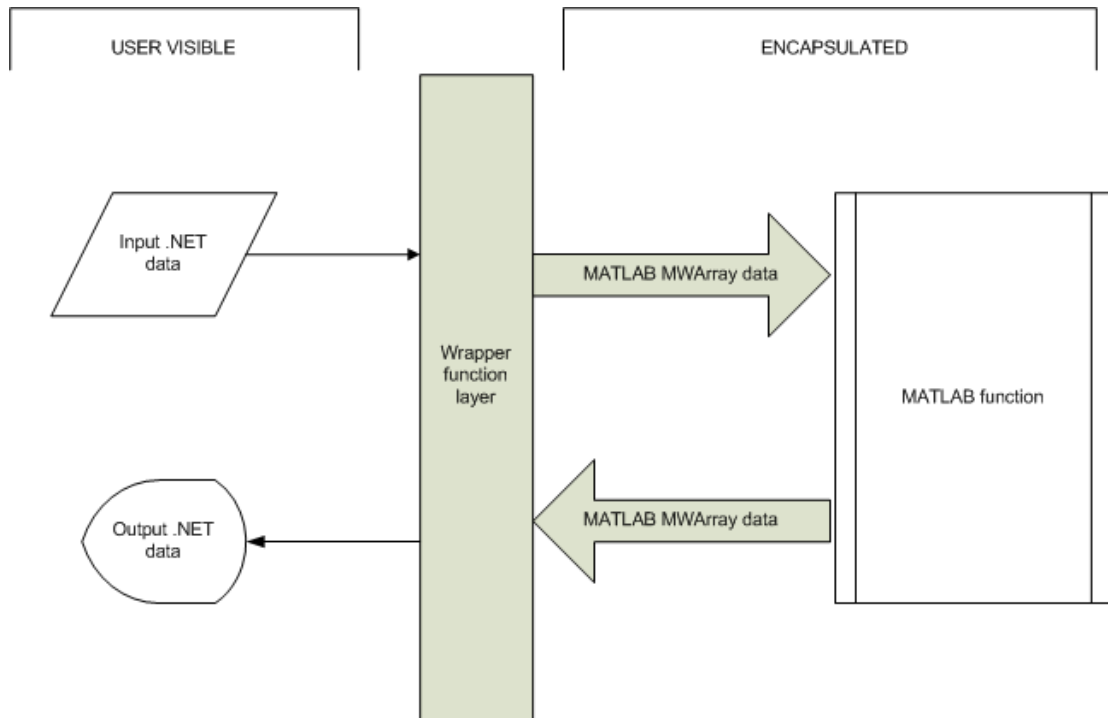
As you can see, manually marshaling data adds complexity and potential failure points to the task of integrating deployed components into a .NET application. This is particularly true for these reasons:

- **Your application cannot detect type mismatch errors until run-time.** For example, you might accidentally create an `MWArray` from a string and pass the array to a deployed function that expects a number. Because the wrapper code generated

by MATLAB Builder NE expects an `MWArray`, the .NET compiler is unable to detect this error and the deployed function either throws an exception or returns the wrong answer.

- **Your end users must learn how to use the `MWArray` data type** or alternately mask the `MWArray` data type behind a manually written (and manually maintained) API. This introduces unwanted training time and places resource demands on a potentially overcommitted staff.

You can avoid performing `MWArray` data marshaling by using *type-safe interfaces*. Such interfaces minimize explicit type conversions by hiding the `MWArray` type from the calling application. Using type-safe interfaces allows .NET Developers to work directly with familiar native data types.



Simplified Data Marshaling With a Type-Safe Interface

Advantages of Implementing a Type-Safe Interface

Some of the reasons to implement type-safe interfaces include:

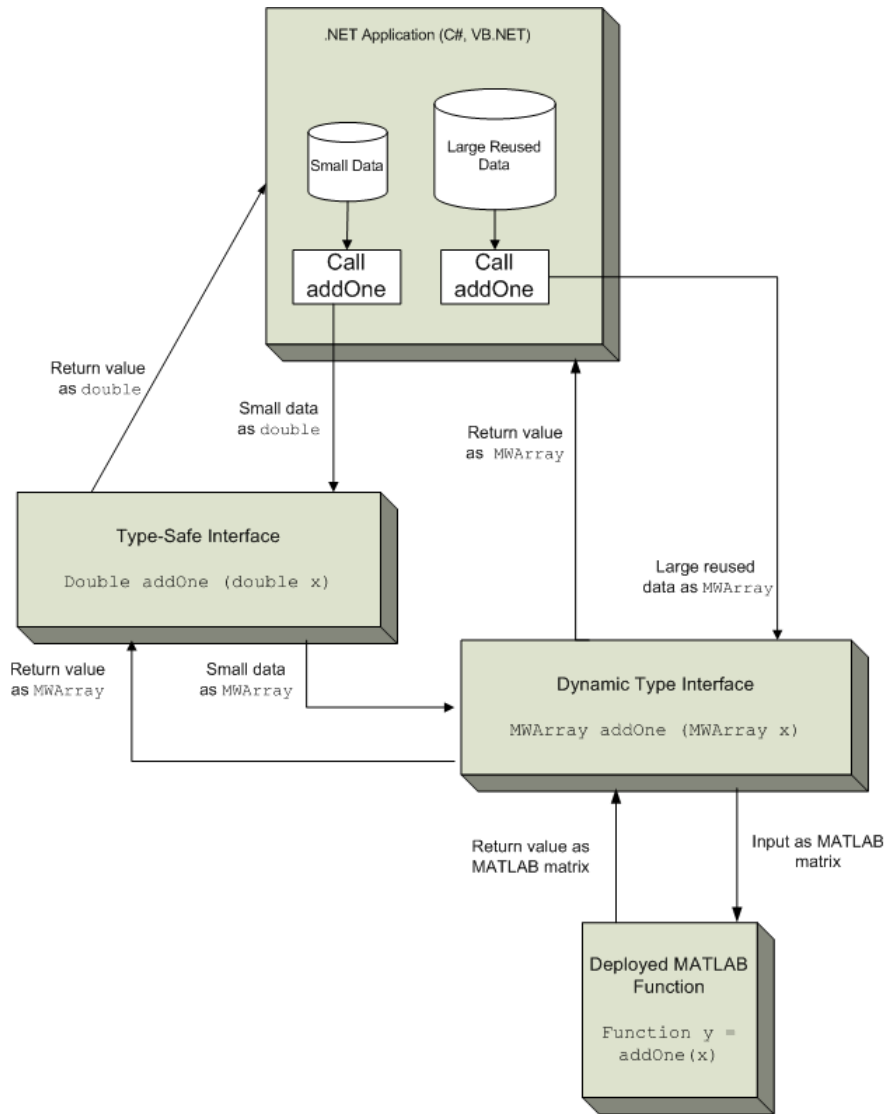
- **You avoid training and coding costs** associated with teaching end users to work with `MWArray`s.
- **You minimize cost of data you must marshal** by either placing `MWArray` objects in type-safe interfaces or by calling `MWArray`-based functions in the deployed component.
- **Flexibility — you mix type-safe interfaces with manual data marshaling** to accommodate data of varying sizes and access patterns. For example, you may have a few large data objects (images, for example) that would incur excess cost to your organization if managed with a type-safe interface. By mixing type-safe interfaces and manual marshaling, smaller data types can be managed automatically with the type-safe interface and your large data can be managed on an as-needed basis.

How Type-Safe Interfaces Work

Every MATLAB Builder NE component exports one or more public methods that accept and return data using `MWArrays`.

Adding a type-safe interface to a MATLAB Builder NE component creates another set of methods (with the same names) that accept and return native .NET types.

The figure Architecture of a Deployed Component with a Type-Safe Interface illustrates the data paths between the .NET host application and the deployed MATLAB function.



Architecture of a Deployed Component with a Type-Safe Interface

The MATLAB function `addOne` returns its input plus one.

Deploying `addOne` with a type-safe interface creates two .NET `addOne` methods:

- One that accepts and returns `.NET double`
- One that accepts and returns `MWArray`.

You may create multiple type-safe interface methods for a single MATLAB function. Type-safe interface methods follow the standard `.NET` methods for overloading.

Notice that the type-safe methods co-exist with the `MWArray`-based methods. Your `.NET` application may mix and match calls to either type of method, as appropriate.

You may find `MWArray` methods more efficient when passing large data values in loops to one or more deployed functions. In such cases, creating an `MWArray` object allows you to marshal the data only once whereas the type-safe interface marshals inputs on every call.

Build Your Component and Generate Your Type-Safe API

In this section...

“Using the Library Compiler” on page 7-8

“Using the Command-Line Tools” on page 7-8

Using the Library Compiler

The Library Compiler generates the type-safe API, when you build your component, if the correct options are selected.

- 1 Create a Library Compiler project.
- 2 Select **.NET Assembly** from the **Type** list.
- 3 Expand the **Additional Runtime Settings** section.
- 4 In the **Type-Safe API** section, do the following:
 - 1 Select **Enable Type-Safe API**.
 - 2 In the **Interface assembly** field, specify the location of the type-safe/WCF interface assembly that you built.
 - 3 Select the desired interface from the **.NET interface** drop-down box.

Tip If the drop-down is blank, the Library Compiler may have been unable to find any .NET interfaces in the assembly you selected.

- 4 Specify the name of the class you want the generated API to wrap, in the **Wrapped Class** field.

Note: Leave the **Namespace** field blank.

- 5 Build the project by clicking the **Package** button.

Using the Command-Line Tools

To generate the type-safe API with your component using `mcc`, do the following:

- 1 Build the component by entering this command from MATLAB:

```
mcc -v -B 'dotnet:AddOneComp,Mechanism,3.5,private,local'  
addOne
```

- See the `mcc` reference page in this user's guide for details on the options specified.
- 2** Generate the type-safe API by entering this command from MATLAB:

```
ntswrap -c AddOneComp.Mechanism -i IAddOne -a IAddOne.dll
```

where:

- `-c` specifies the namespace-qualified name of the MATLAB Builder NE component to wrap with a type-safe API. If the component is scoped to a namespace, specify the full namespace-qualified name (`AddOneComp.Mechanism` in the example). Because no namespace is specified by `ntswrap`, the type-safe interface class appears in the global namespace.
- `-i` specifies the name of the .NET interface that defines the type-safe API. The interface name is usually prefixed by an `I`.
- `-a` specifies the absolute or relative path to the assembly containing the .NET statically-typed interface, referenced by the `-i` switch.

Tip If the assembly containing the .NET interface `IAddOne` is not in the current folder, specify the full path.

Caution Not all arguments are compatible with each other. See the `ntswrap` reference page in this user's guide for details on all command options.

Implementing a Type-Safe Interface

Implementing a type-safe interface usually requires the expertise of a .NET Developer because it requires performing a number of medium-to-advanced programming tasks.

Tip Data objects that merely pass through either the target or MATLAB environments may not need to be marshaled, particularly if they do not cross a process boundary. Because marshaling is costly, only marshal on demand.

After you write and test your MATLAB code, develop a .NET interface that supports the native types through the API in either C# or Visual Basic . In this example, the interface, `IAddOne`, is written in C#.

Each method in the interface must exactly match a deployed MATLAB function.

The `IAddOne` interface specifies six overload of `addOne`:

```
using System.ServiceModel;

[ServiceContract]
public interface IAddOne
{
    [OperationContract(Name = "addOne_1")]
    int addOne(int x);

    [OperationContract(Name = "addOne_2")]
    void addOne(ref int y, int x);

    [OperationContract(Name = "addOne_3")]
    void addOne(int x, ref int y);

    [OperationContract(Name = "addOne_4")]
    System.Double addOne(System.Double x);

    [OperationContract(Name = "addOne_5")]
    System.Double[] addOne(System.Double[] x);

    [OperationContract(Name = "addOne_6")]
    System.Double[][] addOne(System.Double[][] x);
}
```

As you can see, all methods have one input and one output (to match the MATLAB `addOne` function), though the type and position of these parameters varies.

Data Conversion Rules for Using the Type-Safe Interface

- In a MATLAB function, declaration outputs appear before inputs. For example, in the `addOne` function, the output `y` appears before the input `x`. This ordering is not required for .NET interface functions. Inputs may appear before or after outputs or the two may be mixed together.
- MATLAB Builder NE matches .NET interface functions to public MATLAB functions by function name and argument count. In the `addOne` example, both the .NET interface function and the MATLAB function must be named `addOne` and both functions must have an equal number of arguments defined.
- The number and relative order of input and output arguments is critical.
 - In evaluating parameter order, only the order of like parameters (inputs or outputs) is considered, regardless of where they appear in the parameter list.
 - An interface may have fewer inputs than MATLAB functions, but not more.
- Argument mapping occurs according to argument order rather than argument name.
- The function return value, if specified, counts as the first output.
- You must use `out` parameters for multiple outputs.
 - Alternately, the `ref` parameter can be used for `out`. `ref` and `out` parameters are synonymous.
- MATLAB does not support overloading of functions. Thus, all user-supplied overloads of a function with a given name will map to a function (with an identical name) generated by MATLAB Builder NE.

See “.NET Types to MATLAB Types” on page 11-4 for complete guidelines in managing data conversion with type-safe interfaces.

Create Windows Communications Foundation (WCF)[™]-Based Components

In this section...

“What Is WCF?” on page 7-12

“Before Running the WCF Example” on page 7-13

“Deploying a WCF-Based Component” on page 7-13

What Is WCF?

The Windows Communication Foundation[™] (or WCF) is an application programming interface in the .NET Framework for building connected, service-oriented, Web-centric applications.

WCF supports distributed computing using a service-oriented architecture. Clients consume multiple services that can be consumed by multiple clients. Services are loosely coupled to each other.

Services typically have a WSDL interface (Web Services Description Language), which any WCF client can use to consume the service, regardless of which platform the service is hosted on.

A WCF client connects to a WCF service via an *endpoint*. Each service exposes its contract via one or more endpoints. An endpoint has an address, which is a URL specifying where the endpoint can be accessed, and binding properties that specify how the data will be transferred.

What's the Difference Between WCF and .NET Remoting?

You generate native .NET objects using both .NET Remoting (“Creating a Remotable .NET Component”) and native .NET types(WCF).

What's the difference between these two technologies and which should you use?

WCF is an end-to-end *Web Service*. Many of the advantages afforded by .NET Remoting—a wide selection of protocol interoperability, for instance—can be achieved with a WCF interface, in addition to having access to a richer, more flexible set of native data types. .NET Remoting can only support native objects.

WCF offers more robust choices in most every aspect of Web-based development, even implementation of a Java client, for example.

For More information About WCF

For up-to-date information regarding WCF, refer to the MSDN article “Windows Communication Foundation.”

Before Running the WCF Example

Before running this example, keep the following in mind:

- You must be running at least Microsoft .NET Framework 3.5 to use the WCF feature.
- If you want to use WCF, the easiest way to do so is through the type-safe API.
- WCF and .NET Remoting are not compatible in the same deployment project or component.
- The example in this chapter requires both client and server to use message sizes larger than the WCF defaults. For information about changing the default message size, see the MSDN article regarding setting of the `maxreceivedmessagesize` property.

Deploying a WCF-Based Component

Deploying a WCF-based component requires the expertise of a .NET Developer (see description of) because it requires performing a number of advanced programming tasks.

To deploy a WCF-based component, follow this general workflow:

1. “Write and Test Your MATLAB Code” on page 7-14
2. “Develop Your WCF Interface” on page 7-14
3. “Build Your Component and Generate Your Type-Safe API” on page 7-15
4. “Develop Server Program Using the WCF Interface” on page 7-17
5. “Compile the Server Program” on page 7-20
6. “Run the Server Program” on page 7-20
7. “Generate Proxy Code for Clients” on page 7-21
8. “Compile the Client Program” on page 7-22
9. “Run the Client Program” on page 7-24

Write and Test Your MATLAB Code

Create your MATLAB program and then test the code before implementing a type-safe interface. The functions in your MATLAB program must match the declarations in your native .NET interface.

In the following example, the deployable MATLAB code contains one exported function, `addOne`. The `addOne` function adds the value one (1) to the input received. The input must be numeric, either a scalar or a matrix of single or multiple dimensions.

```
function y = addOne(x)
% ADDONE Add one to numeric input. Input must be numeric.

    if ~isnumeric(x)
        error('Input must be numeric. Input was %s.', class(x));
    end
    y = x + 1;
end
```

Note: `addOne` must perform run-time type checking to ensure valid input.

Develop Your WCF Interface

After you write and test your MATLAB code, develop an interface in either C# or Visual Basic that supports the native types through the API.

Define `IAddOne` Overloads

See “Implementing a Type-Safe Interface” on page 7-10 for complete rules on defining interface overloads.

In addition, when using WCF, your overloaded functions *must* have unique names.

Note that in the WCF implementation of `addOne`, you decorate the methods with the `OperationContract` property. You give each method a unique operation name, which you specify with the `Name` property of `OperationContract`, as in this example:

```
using System.ServiceModel;

[ServiceContract]
```

```
public interface IAddOne
{
    [OperationContract(Name = "addOne_1")]
    int addOne(int x);

    [OperationContract(Name = "addOne_2")]
    void addOne(ref int y, int x);

    [OperationContract(Name = "addOne_3")]
    void addOne(int x, ref int y);

    [OperationContract(Name = "addOne_4")]
    System.Double addOne(System.Double x);

    [OperationContract(Name = "addOne_5")]
    System.Double[] addOne(System.Double[] x);

    [OperationContract(Name = "addOne_6")]
    System.Double[][] addOne(System.Double[][] x);
}
```

As you can see, the `IAddOne` interface specifies six overloads of the `addOne` function. Also, notice that all have one input and one output (to match the MATLAB `addOne` function), though the type and position of these parameters varies.

For additional code snippets and data conversion rules regarding type-safe interfaces, see “Implementing a Type-Safe Interface” on page 7-10.

For more information on WCF contracts and properties, see the Microsoft WCF Web Site.

Compile IAddOne into an Assembly

Compile `IAddOne.cs` into an assembly using Microsoft Visual Studio.

Note: This example assumes your assembly contains only `IAddOne`. Realistically, it is more likely that `IAddOne` will already be part of a compiled assembly. The assembly may be complete even before the MATLAB function is written.

Build Your Component and Generate Your Type-Safe API

Use either the Library Compiler or the deployment command line tools to generate the type-safe API.

Using the Library Compiler

The Library Compiler generates the type-safe API, when you build your component, if the correct options are selected.

- 1 Create your Library Compiler project.

When defining your project, use these values:

Project Name	AddOneComp
Class Name	Mechanism
File to compile	addOne

Note: *Do not* click the **Package** button at this time.

- 2 Expand the **Additional Runtime Settings** section.
- 3 On the **Type-Safe API** tab, do the following:
 - a Select **Enable Type-Safe API**.
 - b In the **Interface assembly** field, specify the location of the type-safe/WCF interface assembly that you built.
 - c Select **IAddOne** from the **.NET interface** drop-down box. The interface name is usually prefixed by an **I**.

Tip If the drop-down is blank, the compiler may have been unable to find any **.NET** interfaces in the assembly you selected. Select another assembly.

- d Specify **Mechanism**, as the class name you want the generated API to wrap, in the **Wrapped Class** field.

Note: Leave the **Namespace** field blank.

- 4 Build the project as usual by clicking the **Package** button.

Using the Deployment Command-Line Tools

To generate the type-safe API with your component build (compilation) using `mcc`, do the following:

- 1 Build the component by entering this command from **MATLAB**:

```
mcc -v -B 'dotnet:AddOneComp,Mechanism,3.5,private,local'  
addOne
```

See the `mcc` reference page in this user's guide for details on the options specified.

2 Generate the type-safe API by entering this command from MATLAB:

```
ntswrap -c AddOneComp.Mechanism -i IAddOne -a IAddOne.dll
```

where:

- `-c` specifies the namespace-qualified name of the MATLAB Builder NE component to wrap with a type-safe API. If the component is scoped to a namespace, specify the full namespace-qualified name (`AddOneComp.Mechanism` in the example). Because no namespace is specified by `ntswrap`, the type-safe interface class appears in the global namespace.
- `-i` specifies the name of the .NET interface that defines the type-safe API. The interface name is usually prefixed by an `I`.
- `-a` specifies the absolute or relative path to the assembly containing the .NET statically-typed interface, referenced by the `-i` switch.

Tip If the assembly containing the .NET interface `IAddOne` is not in the current folder, specify the full path.

Caution Not all arguments are compatible with each other. See the `ntswrap` reference page in this user's guide for details on all command options.

Develop Server Program Using the WCF Interface

You have now built your component and generated a WCF-compliant type-safe API.

Next, develop a server program that provides access (via the `WCFServiceContract`) to the overloads of `addOne` defined by the WCF `IAddOne` interface. The program references an `App.config` XML configuration file.

The WCF server program loads the WCF-based `addOne.Mechanism` component and makes it available to SOAP clients via the type-safe `mechanismIAddOne` interface.

About Jagged Array Processing When writing your interface, you will be coding to handle jagged arrays, as opposed to rectangular arrays. For more information about jagged arrays, see “Jagged Array Processing” on page 5-22 in this documentation.

WCF Server Program

```
using System;
using System.Text;
using System.ServiceModel;

namespace AddMasterServer
{
    class AddMasterServer
    {
        static void Main(string[] args)
        {
            try
            {
                using (ServiceHost host =
                    new ServiceHost(typeof(MechanismIAddOne)))
                {
                    host.Open();
                    Console.WriteLine("
                        AddMaster Server is up running.....");
                    Console.WriteLine("
                        Press any key to close the service.");
                    Console.ReadLine();
                    Console.WriteLine("Closing service...");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

App.config XML file

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration=
        "AddMaster.ServiceBehavior" name="MechanismIAddOne">
        <endpoint
          address=""
          binding="wsHttpBinding"
          contract="IAddOne"
          name="HttpBinding" />
        <endpoint
          address=""
          binding="netTcpBinding"
          contract="IAddOne"
          name="netTcpBinding" />
        <endpoint
          address="mex"
          binding="mexHttpBinding"
          contract="IMetadataExchange"
          name="MexHttpBinding" />
        <endpoint
          address="mex"
          binding="mexTcpBinding"
          contract="IMetadataExchange"
          name="MexTCPBinding" />
        <host>
          <baseAddresses>
            <add baseAddress=
              "http://localhost:8001/AddMaster/" />
            <add baseAddress=
              "net.tcp://localhost:8002/AddMaster/" />
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="AddMaster.ServiceBehavior">
```

```
<serviceMetadata httpGetEnabled="True" httpGetUrl=
    "http://localhost:8001/AddMaster/mex" />
<!-- To receive exception details in faults for
<!-- debugging purposes,
set the value below to true. Set to false before
deployment to avoid disclosing exception
information -->
    <serviceDebug includeExceptionDetailInFaults="True" />
</behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>
```

Compile the Server Program

Compile the server program using Microsoft Visual Studio by doing the following:

- 1 Create a Microsoft Visual Studio project named `AddMaster`.
- 2 Add `AddMasterServer.cs` and `App.config` (the configuration file created in the previous step) to your project.
- 3 Add references in the project to the following files.

This reference:	Defines:
<code>IAddOne.dll</code>	The .NET native type interface <code>IAddOne</code>
<code>MechanismIAddOne.dll</code>	The generated type-safe API
<code>AddOneCompNative.dll</code>	The MATLAB Builder NE component

Note: Unlike other .NET deployment scenarios, you do not need to reference `MWArray.dll` in the server program source code. The `MWArray` data types are hidden behind the type-safe API in `MechanismIAddOne`.

- 4 If you are not already referencing `System.ServiceModel`, add it to your Visual Studio project.
- 5 Compile the program with Microsoft Visual Studio.

Run the Server Program

Run the server program from a command line.

The output should look similar to the following.

AddMaster Server is up running.....
Press any key to close the service.

Pressing a key results in the following.

Closing service....

Generate Proxy Code for Clients

Configure your clients to communicate with the server by running the automatic proxy generation tool, `svcutil.exe`. Most versions of Microsoft Visual Studio can automatically generate client proxy code from server metadata.

Caution Before you generate your client proxy code using this step, the server *must* be available and running. Otherwise, the client will not find the server.

- 1 Create a client project in Microsoft Visual Studio.
- 2 Add references by using either of these two methods. See “Port Reservations and Using localhost 8001” on page 7-22 for information about modifying port configurations.

Method 1	Method 2
<p>a In the Solutions Explorer pane, right-click References.</p> <p>b Select Add Service Reference. The Add Service Reference dialog box appears.</p> <p>c In the Address field, enter: <code>http://localhost:8001/AddMaster/</code></p> <hr/> <p>Note: Be sure to include the / following <code>AddMaster</code>.</p> <p>d In the Namespace field, enter <code>AddMasterProxy</code>.</p> <p>e Click OK.</p>	<p>a Enter the following command from your client application directory to generate <code>AddMasterProxy.cs</code>, which contains client proxy code. This command also generates configuration file <code>App.config</code>.</p> <pre>svcutil.exe /t:code http://localhost:8001/AddMaster/ /out:AddMasterProxy.cs /config:App.config</pre> <hr/> <p>Note: Enter the above command on one line, without breaks.</p> <p>b Add <code>AddMasterProxy.cs</code> and <code>App.config</code> to your client project</p>

Port Reservations and Using localhost 8001

When running a self-hosted application, you may encounter issues with port reservations. Use one of the tools below to modify your port configurations, as necessary.

if You Run....	Use This Tool to Modify Port Configurations....
Windows XP	httpcfg
Windows Vista™	netsh
Windows 7	netsh

Compile the Client Program

The client program differs from the `AddMaster.cs` server program as follows:

- At start-up, this program connects to the `AddMasterService` provided by the `AddMaster` WCF service.
- Instead of directly invoking the methods of the type-safe mechanism `IAddOne` interface, the WCF client uses the method names defined in the `OperationContract` attributes of `IAddOne`.

Compile the client program by doing the following:

- 1 Add the client code (`AddMasterClient.cs`) to your Microsoft Visual Studio project.
- 2 If you are not already referencing `System.ServiceModel`, add it to your Visual Studio project.
- 3 Compile the WCF client program in Visual Studio.

WCF Client Program

```
using System;
using System.Text;
using System.ServiceModel;

namespace AddMasterClient
{
    class AddMasterClient
    {
        static void Main(string[] args)
        {
            try
```

```
{
    // Connect to AddMaster Service
    Console.WriteLine("Connecting to
        AddMaster Service through
        Http connection...");
    AddOneClient AddMaster =
        new AddOneClient("HttpBinding");
    Console.WriteLine("Connected to
        AddMaster Service...");

    // Output as return value
    int one = 1;
    int two = AddMaster.addOne_1(one);
    Console.WriteLine("addOne({0}) = {1}",
        one, two);

    // Output: first parameter
    int i16 = 16;
    int o17 = 0;
    AddMaster.addOne_2(ref o17, i16);
    Console.WriteLine("addOne({0}) = {1}",
        i16, o17);

    // Output: second parameter
    int three = 0;
    AddMaster.addOne_3(two, ref three);
    Console.WriteLine("addOne({0}) = {1}",
        two, three);

    // Scalar doubles
    System.Double i495 = 495.0;
    System.Double third =
        AddMaster.addOne_4(i495);
    Console.WriteLine("addOne({0}) = {1}",
        i495, third);

    // Vector addition
    System.Double[] i = { 30, 60, 88 };
    System.Double[] o = AddMaster.addOne_5(i);
    Console.WriteLine(
        "addOne([ {0} {1} {2} ]) = [ {3} {4} {5} ]",
        i[0], i[1], i[2], o[0], o[1], o[2]);
}
catch (Exception ex)
```

```
        {  
            Console.WriteLine(ex.Message);  
        }  
  
        Console.WriteLine("Press any key to close  
                           the client application.");  
        Console.ReadLine();  
        Console.WriteLine("Closing client...");  
    }  
}
```

Run the Client Program

Run the client program from a command line.

The output should be similar to the following:

```
Conncting to AddMaster Service through Http connection...  
Connected to AddMaster Service...  
addOne(1) = 2  
addOne(16) = 17  
addOne(2) = 3  
addOne(495) = 496  
addOne([30 60 88]) = [31 61 89]  
addOne([0 2; 3 1]) = [1 3; 4 2]  
Press any key to close the client application.
```

Pressing a key results in the following.

```
Closing client....
```

Create Managed Extensibility Framework (MEF) Plug-Ins

In this section...

“What Is MEF?” on page 7-25

“MEF Prerequisites” on page 7-26

“Addition and Multiplication Applications with MEF” on page 7-26

What Is MEF?

The Managed Extensibility Framework (MEF) is a library for creating lightweight, extensible applications.

Why Use MEF?

When working with .NET applications, it is typically necessary to specify which .NET components should be loaded.

Keeping the application updated with hard-coded names and locations of .NET components rapidly becomes a maintenance issue, especially if the updating is to be done by an end user who may not be familiar with the technical aspects of the application.

MEF allows you to create a *plug-in* framework for your application or use an existing framework with no required preconfiguration. It lets you avoid hard-coded dependencies and reuse extensions within and across applications. Using MEF lets you avoid recompiling applications, such as Microsoft Silverlight™, for which source code is generally unavailable.

How Does MEF Work?

MEF provides a way for .NET components to be automatically discovered. It does this by using MEF components called *parts*. Parts declaratively specify *dependencies* (imports) and *capabilities* (exports) through metadata.

An MEF application consists of a *host* program that invokes functions defined in MEF parts. MEF Parts that implement the same interface export functions with identical names. These parts all participate in a common framework.

Each part implements an interface; often times, many parts implement the same interface. Parts that implement the same interface export functions with identical names that can be used over a variety of applications. MEF parts that implement the same interface must have descriptive, unique metadata.

The MEF host examines each part's metadata to determine which to load and invoke.

MEF parts are similar to MATLAB MEX files—each MEX file dynamically extends MATLAB just as parts dynamically extend .NET components.

For More information About MEF

For up-to-date information regarding MEF, refer to the MSDN article “Managed Extensibility Framework.”

MEF Prerequisites

Before running this example, keep the following in mind:

- You must be running Microsoft Visual Studio 2010 to create MEF applications. If you can't use Visual Studio 2010, you can't run this example code, or any other program that uses MEF. End Users do not need Microsoft Visual Studio 2010 to run applications using MEF.
- You must be running at least Microsoft .NET Framework 4.0 to use the MEF feature.
- If you want to use MEF, the easiest way to do so is through the type-safe API.

Addition and Multiplication Applications with MEF

This MEF example application consists of an MEF host and two parts. The parts implement a very simple interface (`ICompute`) which defines three overloads of a single function (`compute`).

Each part performs simple arithmetic. In one part, the `compute` function adds one (1) to its input. In the other part, `compute` multiplies its input by two (2). The MEF host loads both parts and calls their `compute` functions twice.

To run this example, you'll create a new solution containing three projects:

- MEF host
- Contract interface assembly
- Strongly-typed metadata attribute assembly

Implementing MEF requires the expertise of a .NET Developer (see description of) because it requires performing a number of advanced programming tasks.

Where To Find Example Code for MEF

Selected example code can be found, along with some Microsoft Visual Studio projects, in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET`. This code has been tested to be compliant with Microsoft Visual Studio 2010 running on Microsoft .NET Framework version 4.0 or higher.

To deploy an MEF-based component, follow this general workflow:

1. “Create an MEFHost Assembly” on page 7-27
2. “Create a Contract Interface Assembly” on page 7-28
3. “Create a Metadata Attribute Assembly” on page 7-29
4. “Add Contract and Attributes References to MEFHost” on page 7-30
5. “Compile Your Code in Microsoft Visual Studio” on page 7-30
6. “Write MATLAB Functions for MEF Parts” on page 7-30
7. “Create Metadata Files” on page 7-31
8. “Build .NET Components from MATLAB Functions and Metadata” on page 7-31
9. “Install MEF Parts” on page 7-32
10. “Run the MEF Host Program” on page 7-33

Create an MEFHost Assembly

- 1 Start Microsoft Visual Studio 2010.
- 2 Click **File > New > Project**.
- 3 In the **Installed Templates** pane, click **Visual C#** to filter the list of available templates.
- 4 Select the **Console Application** template from the list.
- 5 In the **Name** field, enter **MEFHost**.
- 6 Click **OK**. Your project is created.
- 7 Replace the contents of the default **Program.cs** with the **MEFHost.cs** code. For information about locating example code, see “Where to Find Example Code,” above.
- 8 In the **Solution Explorer** pane, select the project **MEFHost** and right-click. Select **Add Reference**.
- 9 Navigate to the **.NET** tab and add a reference to **System.ComponentModel.Composition**.

- 10** To prevent security errors, particularly if you have a non-local installation of MATLAB, add an application configuration file to the project. This XML file instructs the MEF host to trust assemblies loaded from the network. If your project does not include this configuration file, your application fails at runtime.
 - a** Select the **MEFHost** project in the **Solution Explorer** pane and right-click.
 - b** Click **Add > New Item**.
 - c** From the list of available items, select **Application Configuration File**.
 - d** Click **Add**. The configuration file is added to your project. Visual Studio automatically names the file **App.config**.
 - e** Replace the automatically-generated contents of **App.config** with this configuration:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <loadFromRemoteSources enabled="true" />
  </runtime>
</configuration>
```

You have finished building the first project, which builds the MEF host.

Next, you add a C# class library project for the MEF contract interface assembly.

Create a Contract Interface Assembly

- 1** in Visual Studio, click **File > New > Project**.
- 2** In the **Installed Templates** pane, click **Visual C#** to filter the list of available templates.
- 3** Select the **Class Library** template from the list.
- 4** In the **Name** field, enter **Contract**.

Note: Ensure **Add to solution** is selected in the **Solution** drop-down box.

- 5** Click **OK**. Your project is created.
- 6** Replace the contents of the default **Class1.cs** with the following **ICompute** interface code:

```
namespace Contract
```



```

{
    public interface ICompute
    {
        double compute(double y);
        double[] compute(double[] y);
        double[,] compute(double[,] y);
    }
}

```

You have finished building the second project, which builds the Contract Interface Assembly.

Since strongly-typed metadata requires that you decorate MEF parts with a custom metadata attribute, in the next step you add a C# class library project. This project builds an attribute assembly to your MEFHost solution.

Create a Metadata Attribute Assembly

- 1 In Visual Studio, click **File > New > Project**.
- 2 In the **Installed Templates** pane, click **Visual C#** to filter the list of available templates.
- 3 Select the **Class Library** template from the list.
- 4 In the **Name** field, enter **Attribute**.

Note: Ensure **Add to solution** is selected in the **Solution** drop-down box.

- 5 Click **OK**. Your project is created.
- 6 In the generated assembly code, change the namespace from **Attribute** to **MEFHost**. Your namespace code should now look like the following:

```

namespace MEFHost
{
    public class Class1
    {
    }
}

```

- 7 In the **MEFHost** namespace, replace the contents of the default class **Class1.cs** with the following code for the **ComputationTypeAttribute** class:

```
using System.ComponentModel.Composition;
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public class ComputationTypeAttribute: ExportAttribute
{
    public ComputationTypeAttribute() :
        base(typeof(Contract.ICompute)) { }
    public Operation FunctionType{ get; set; }
    public double Operand { get; set; }
}

public enum Operation
{
    Plus,
    Times
}
```

- 8 Navigate to the **.NET** tab and add a reference to `System.ComponentModel.Composition.dll`.

Add Contract and Attributes References to MEFHost

Before compiling your code in Microsoft Visual Studio:

- 1 In your **MEFHost** project, add references to the **Contract** and **Attribute** projects.
- 2 In your **Attribute** project, add a reference to the **Contract** project.

Compile Your Code in Microsoft Visual Studio

Build all your code by selecting the solution name **MEFHost** in the **Solution Explorer** pane, right-clicking, and selecting **Build Solution**.

In doing so, you create the following binaries in `MEFHost/bin/Debug`:

- `Attribute.dll`
- `Contract.dll`
- `MEFHost.exe`

Write MATLAB Functions for MEF Parts

Create two MATLAB functions. Each must be named `compute` and stored in separate folders, within your Microsoft Visual Studio project:

MEFHost/Multiply/compute.m

```
function y = compute(x)
    y = x * 2;
```

MEFHost/Add/compute.m

```
function y = compute(x)
    y = x + 1;
```

Create Metadata Files

Create a metadata file for each MATLAB function.

1 For MEFHost/Add/compute.m:

a Name the metadata file MEFHost/Add/Add.metadata.

b In this file, enter the following metadata on one line:

```
[MEFHost.ComputationType(FunctionType=MEFHost.Operation.Plus, Operand=1)]
```

2 For MEFHost/Multiply/compute.m:

a Name the metadata file MEFHost/Multiply/Multiply.metadata.

b In this file, enter the following metadata on one line:

```
[MEFHost.ComputationType(FunctionType=MEFHost.Operation.Times, Operand=2)]
```

Build .NET Components from MATLAB Functions and Metadata

In this step, use the **Library Compiler** app to create .NET components from the MATLAB functions and associated metadata.

Use the information in these tables to create both **Addition** and **Multiplication** projects.


Note: Since you are deploying two functions, you need to run the **Library Compiler** app *twice*, once using the **Addition.prj** information and once using the following **Multiplication.prj** information.

Addition.prj

Project Name	Addition
Class Name	Add
File to compile	MEFHost/Add/compute.m

Multiplication.prj

Project Name	Multiplication
Class Name	Multiply
File to compile	MEFHost/Multiply/compute.m

- 1 Click the **Library Compiler** app in the apps gallery.
- 2 Create your component, following the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11.
- 3 Modify project settings ( > **Settings**) on the **Type Safe API** tab, for whatever project you are building (**Addition** or **Multiplication**).

Project Setting	Addition.prj	Multiplication.prj
Enable Type Safe API	Checked	Checked
Interface Assembly	MEFHost/bin/Debug/Contract.dll	MEFHost/bin/Debug/Contract.dll
MEF metadata	MEFHost/Add/Add.metadata	MEFHost/Multiply/Multiply.metadata
Attribute Assembly	MEFHost/bin/Debug/Attribute.dll	MEFHost/bin/Debug/Attribute.dll
Wrapped Class	Add	Multiply

- 4 Click the Package button.

Install MEF Parts

The two components you have built are MEF parts. You now need to move the generated parts into the catalog directory so your application can find them:

- 1 Create a parts folder named MEFHost/Parts.

- 2 If necessary, modify the path argument that is passed to the `DirectoryCatalog` constructor in your MEF host program. It must match the full path to the `Parts` folder that you just created.

Note: If you change the path after building the MEF host a first time, you must rebuild the MEF host again to pick up the new `Parts` path.

- 3 Copy the two `componentNative.dlls` (`Addition` and `Multiplication`) and `AddICompute.dll` and `MultiplyICompute.dll` assemblies from your into `MEFHost/Parts`.

Note: You do not need to reference any of your MEF part assemblies in the MEF host program. The host program uses a `DirectoryCatalog`, which means it automatically searches for (and loads) parts that it finds in the specified folder. You can add parts at any time, without having to recompile or relink the MEF host application. You do not need to copy `Addition.dll` or `Multiplication.dll` to the `Parts` directory.

Run the MEF Host Program

MATLAB-based MEF parts require the MATLAB runtime, like all deployed MATLAB code.

Before you run your MEF host, ensure that the correct version of the MATLAB runtime is available and that `matlabroot/runtime/arch` is on your path.

- 1 From a command window, run the following. This example assumes you are running from `c:\Work`.

```
c:\Work> MEFHost\bin\Debug\MEFHost.exe
```

- 2 Verify you receive the following output:

```
8 Plus 1 = 9
9 Times 2 = 18
16 Plus 1 = 17
1.5707963267949 Times 2 = 3.14159265358979
```

Troubleshooting the MEF Host Program

Do you receive an exception indicating that a type initializer failed?

Ensure that you:

- Have `matlabroot/runtime/arch` defined to your MATLAB path.
- Have .NET security permissions set to allow applications to load assemblies from a network.
- Rebuilt MEFHost after adding the application configuration file.

Do you receive an exception indicating that `MWArray.dll` cannot be loaded commonly?

Ensure that you:

- Installed `MWArray.dll` in the Global Assembly Cache (GAC).
- Match the bit-depth of `MWArray.dll` to the bit depth of your MEF host application.

Often the default architecture for a C# console application is 32 bits. If you've installed the 64-bit version of `MWArray.dll` into the GAC, you'll get this error. The easiest correction for this error is to change your console application to 64-bit. To do this in Microsoft Visual Studio, set **Properties > Build > Platform Target** to **x64**.

Do you receive an exception that a particular version of `mclmcrnt` cannot load?

Ensure that you:

- Do not have more than one instance of MATLAB on your path or installed on your system.
- Have the correct version of `MWArray.dll` installed in the Global Assembly Cache (GAC).

Web Deployment of Figures and Images

- “WebFigures” on page 8-2
- “Create and Modify a MATLAB Figure” on page 8-26
- “Working with MATLAB Figure and Image Data ” on page 8-29

WebFigures

In this section...

“Supported Renderers for WebFigures” on page 8-2
“WebFigures Prerequisites” on page 8-2
“Quick Start Implementation of WebFigures” on page 8-4
“Advanced Configuration of a WebFigure” on page 8-10
“Upgrading Your WebFigures” on page 8-23
“Troubleshooting” on page 8-24
“Logging Levels” on page 8-25

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 Implementation of WebFigures” on page 8-4, 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 the MATLAB renderer.

For more information about MATLAB renderers, see the MATLAB documentation.

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.

WebFigures Prerequisites

- “What You Need to Know to Implement WebFigures” on page 8-3

- “Required Products” on page 8-3
- “Assumptions About the Examples” on page 8-3

What You Need to Know to Implement WebFigures

The following knowledge is assumed when you implement WebFigures for .NET:

- 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, 2008 or 2010
MATLAB Compiler	Microsoft .NET Framework 2.0 or later
MATLAB Builder NE	MATLAB 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 with a namespace of `MyComponent` and a class of `MyComponentclass`.
- Assume the MATLAB runtime has been installed. If not, refer to “Distributing MATLAB Code Using the MATLAB Runtime” 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 8-10 to work with WebFigures unless you are deploying a Web site which is 32-bit only and you have a 32-bit MATLAB runtime installed.

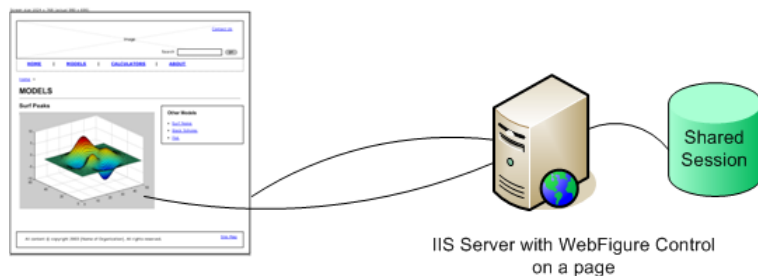
Quick Start Implementation of WebFigures

- “Overview” on page 8-4
- “Procedure” on page 8-4

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 “WebFigures” on page 8-2.

- 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 performing these tasks.

- a In your Visual Studio project, add a reference to *matlabroot*\toolbox\dotnetbuilder\bin\arch\v2.0\WebFiguresService.dll, (where *matlabroot* is the location of the installed MATLAB runtime for machines with an installed MATLAB runtime and *matlabroot* on a MATLAB Builder NE development machine without the MATLAB runtime installed).

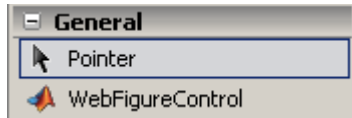
Note: If you are running on a system with 64-bit architecture, use the information in “Advanced Configuration of a WebFigure” on page 8-10 to work with WebFigures unless you are deploying a Web site which is 32-bit only and you have a 32-bit MATLAB runtime installed.

- b Add the following HTML to display the control on the Web application toolbar:

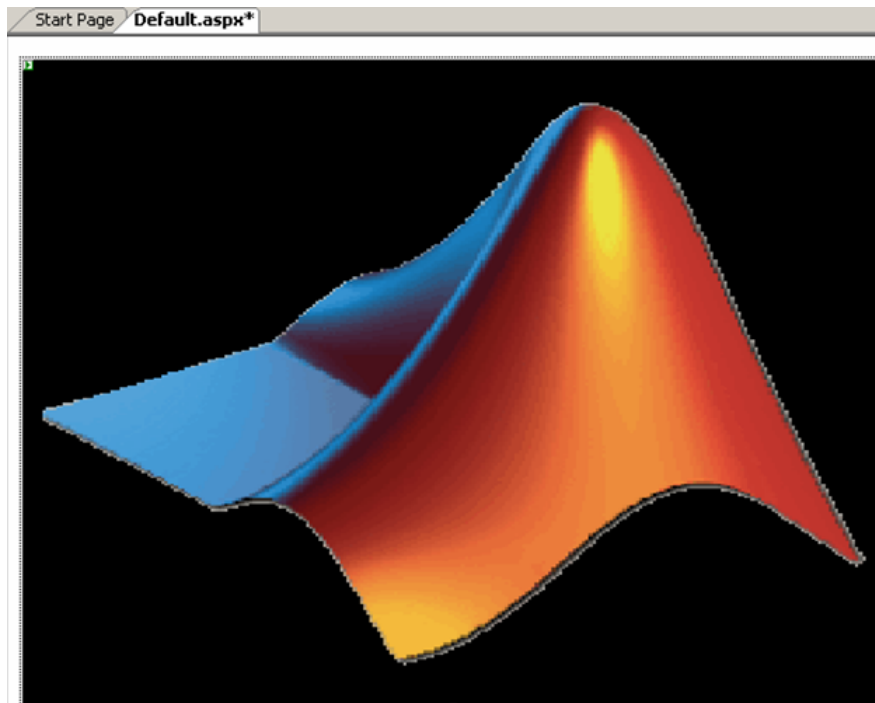
```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="Default" %>
<%@ Register assembly="WebFiguresService, Version=WebFiguresVersion, Culture=neutral, PublicKeyToken=39595b241e56fb4a" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <cc1:webfigurecontrol ID="WebFigureControl1" runat="server" />
    </div>
  </form>
</body>
</html>
```

WebFiguresVersion is the version number of the WebFigure service you are using.

Once `WebFiguresService.dll` is referenced and the HTML added, you will see the following `WebFigureControl` in the **General** section of the Microsoft Visual Studio toolbar:

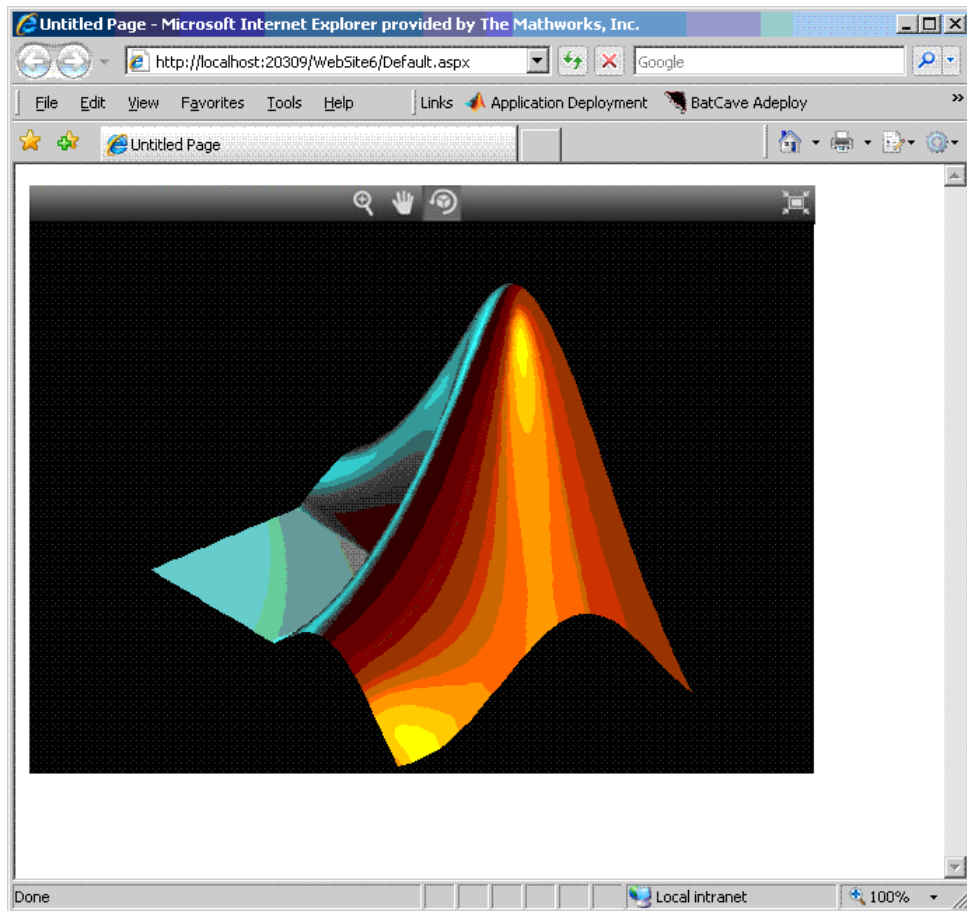


- 5 Drag the `WebFigureControl` from the toolbar to your Web page. After dragging, the Web page displays the following default figure.



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 8-3). See “Common Integration Tasks” on page 5-2 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

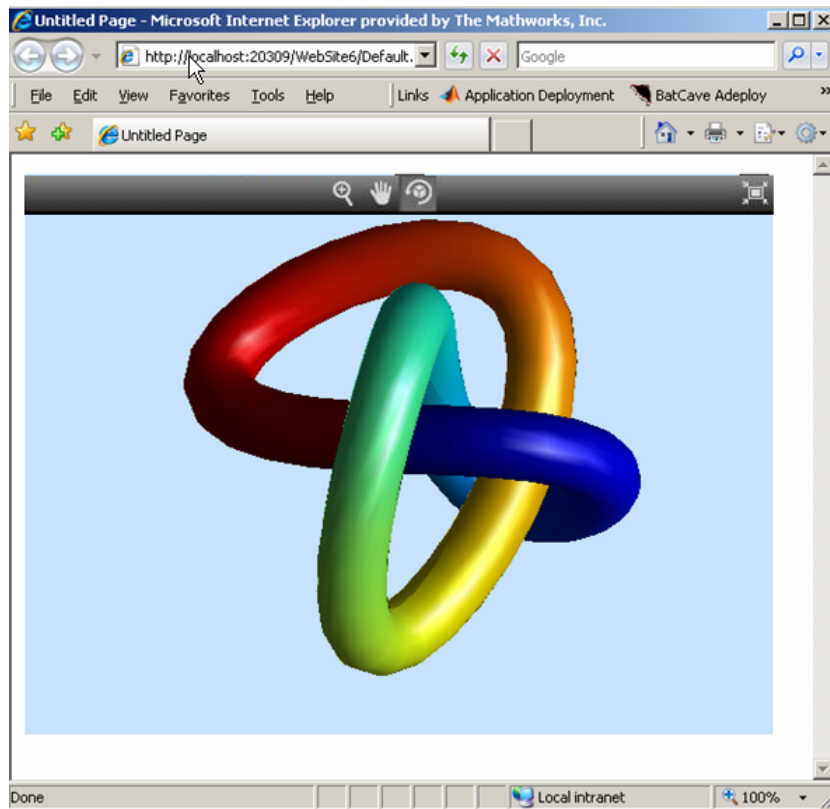
Protected Sub Page_Load(ByVal sender As Object,
                        ByVal e As System.EventArgs)
    Handles Me.Load
    Dim myDeployedComponent As _
        New MyComponentclass()
    WebFigureControl1.WebFigure = _
        New WebFigure(myDeployedComponent.getKnot())
End Sub
```

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 Assembly Cache (Global.asax) to Create WebFigures at Server Start-Up” on page 8-22.

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 8-10 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.



Advanced Configuration of a WebFigure

- “Overview” on page 8-10
- “Manually Installing WebFigureService” on page 8-12
- “Retrieving Multiple WebFigures From a Component” on page 8-14
- “Attaching a WebFigure” on page 8-16
- “Setting Up WebFigureControl for Remote Invocation” on page 8-18
- “Getting an Embeddable String That References a WebFigure Attached to a WebFigureService” on page 8-20
- “Improving Processing Times for JavaScript Using Minification” on page 8-21
- “Using Global Assembly Cache (Global.asax) to Create WebFigures at Server Start-Up” on page 8-22

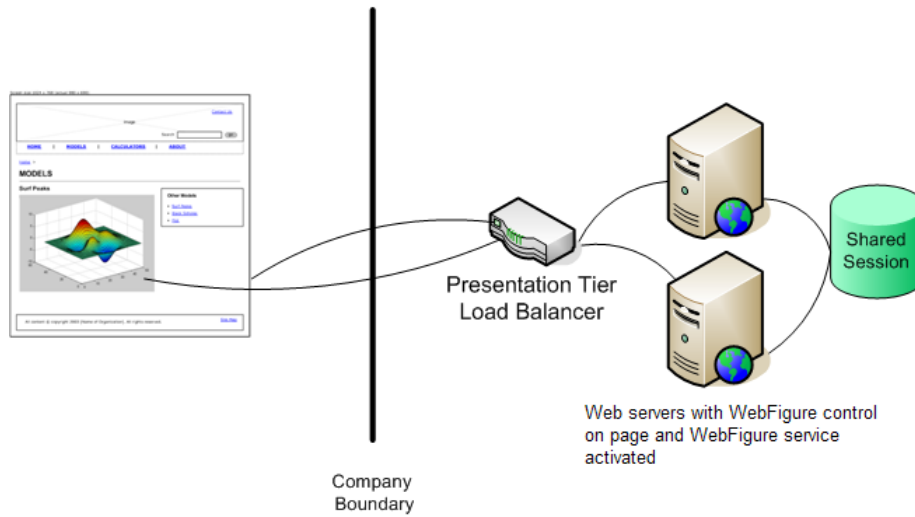
Overview

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 MATLAB runtime (for performance reasons) or place customer-sensitive customer data behind a security firewall, if needed.

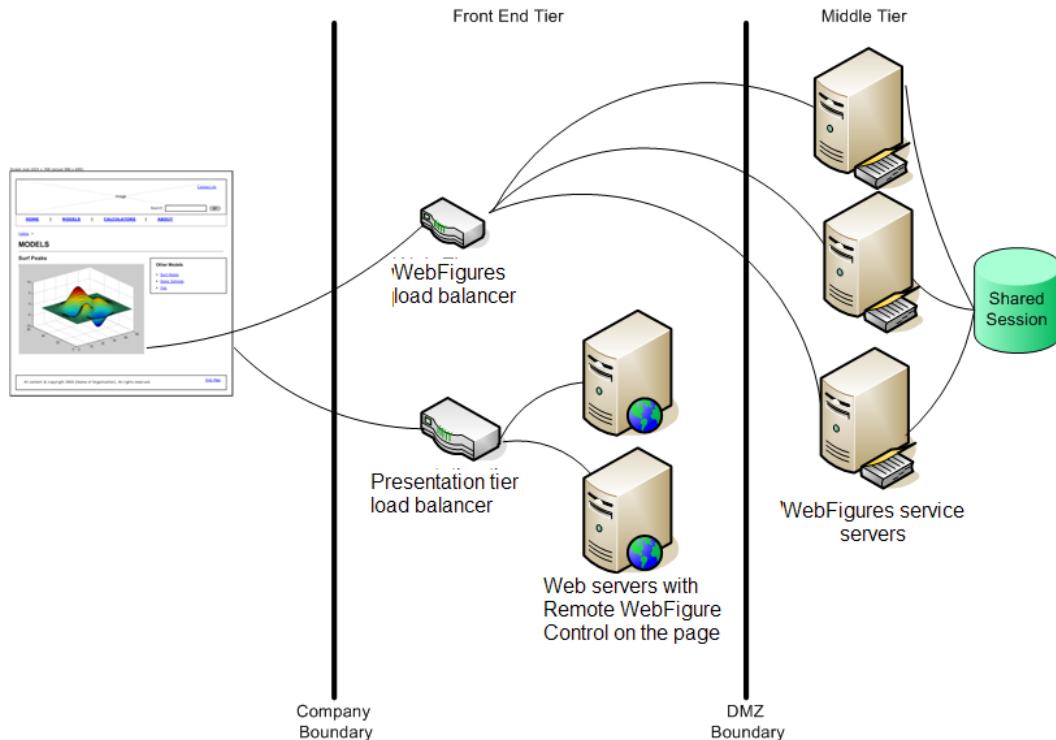
In summary, the advanced configuration offers more choices and adaptability for the user more familiar with Web environments and related technology, as illustrated by the following graphics.

This section describes various ways to customize the basic WebFigures implementation described in “Quick Start Implementation of WebFigures” on page 8-4.

Using the WebFigure Control on the Frontend Servers Outside the Firewall



Using the Remote WebFigure Control to Keep the WebFigure Service Cluster Behind the Firewall



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 Implementation of WebFigures” on page 8-4.
- 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 Implementation of WebFigures” on page 8-4, 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 `matlabroot\toolbox\dotnetbuilder\bin\arch\v2.0` to the project, (where `matlabroot` is the location of the installed MATLAB runtime for machines with an installed MATLAB runtime and `matlabroot` on a MATLAB Builder NE development machine without the MATLAB runtime installed).
- 2 Add the following code to `web.config`. This code tells IIS to send any requests that come to the `__WebFigures.ashx` file to the `WebFigureHttpHandlerFactory` in the `WebFiguresService.dll`:

For Versions of IIS Before 7.0

```
<httpHandlers>
  <add path="__WebFigures.ashx"
        verb="GET"
        type="MathWorks.MATLAB.NET.WebFigures.
            Service.Handlers.Factories.
            Http.WebFigureHttpHandlerFactory"
        validate="false" />
</httpHandlers>
```

For IIS 7.0

```
<system.webServer>
  <handlers>
    <add name="WebFigures" path="__WebFigures.ashx"
          verb="GET"
          type="MathWorks.MATLAB.NET.WebFigures.
              Service.Handlers.Factories.
              Http.WebFigureHttpHandlerFactory"/>
  </handlers>
</system.webServer>
```

Note: The value for the `type=` statement in the above code *must be entered on one continuous line* even though it is not represented as such in the documentation.

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 MArray() = _
            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 <code>WebFigureControl</code> 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 <code>Application</code> , 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 Assembly Cache (`Global.asax`) to Create `WebFigures` at Server Start-Up” on page 8-22.

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 Implementation of WebFigures” on page 8-4, 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 Implementation of WebFigures” on page 8-4.

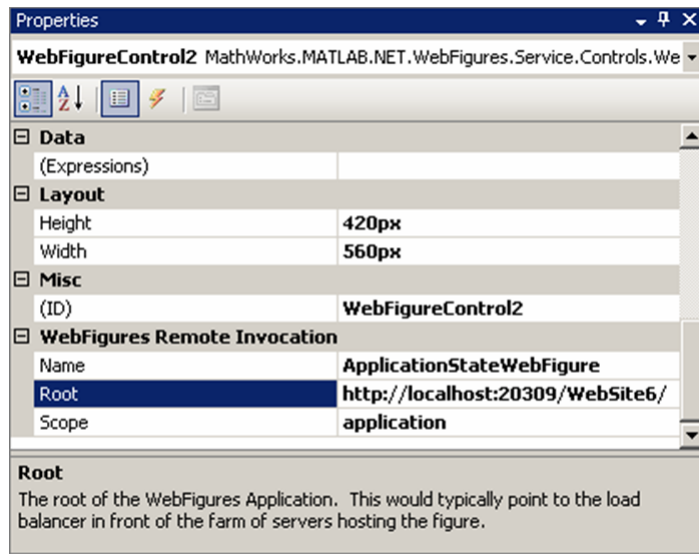
Note: If you are running on a system with 64-bit architecture, use the information in “Advanced Configuration of a WebFigure” on page 8-10 to work with WebFigures

unless you are deploying a Web site which is 32-bit only and you have a 32-bit MATLAB runtime 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 8-12 in “Attaching a WebFigure” on page 8-16.

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 8-16.

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 <code>Session</code> state, then <code>Application</code> state, then <code>Cache</code> 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 Assembly Cache (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 Assembly Cache*, 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 Implementation of WebFigures” on page 8-4, 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 8-16 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 8-18 or directly from the Web page as in “Getting an Embeddable String That References a `WebFigure` Attached to a `WebFigureService`” on page 8-20.

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:

- MATLAB runtime 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 MATLAB runtime.
- 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 8-12.
- `WebFigure` is not attached to `WebFigureService`. See “Attaching a `WebFigure`” on page 8-16.
- 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 <code>WebFigure</code> has been deleted. Refreshing the session will reestablish the <code>WebFigure</code> in cache and the figure will reappear.
No <code>WebFigure</code> Can Be Found with the Name Specified	The <code>WebFigure</code> isn't attached correctly. See “Attaching a <code>WebFigure</code> ” on page 8-16.

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 httpHandlerFactory could not be found on the server specified. See “Manually Installing WebFigureService” on page 8-12.
Could not find a part of the path <i>pathname</i>	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 `yourwebappnameWFSLog.txt`.

Create and Modify a MATLAB Figure

In this section...

“Preparing a MATLAB Figure for Export” on page 8-26

“Changing the Figure (Optional)” on page 8-26

“Exporting the Figure” on page 8-27

“Cleaning Up the Figure Window” on page 8-27

“Modify and Export Figure Data” on page 8-27

MATLAB Programmer

Role	Knowledge Base	Responsibilities
MATLAB programmer	<ul style="list-style-type: none"> • MATLAB expert • No IT experience • No access to IT systems 	<ul style="list-style-type: none"> • Develops models; implements in MATLAB • 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);
```

Modify and Export 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 8-29

“Work with Figures” on page 8-29

“Work with Images” on page 8-30

Front-End Web Developer

Role	Knowledge Base	Responsibilities
Front-end Web developer	<ul style="list-style-type: none"> • No MATLAB experience • Minimal IT experience • Expert at usability and Web page design • Minimal access to IT systems • Expert at ASPX 	<ul style="list-style-type: none"> • As service consumer, manages presentation and usability • Creates front-end applications • Integrates MATLAB code with language-specific frameworks and environments • 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 “*The MATLAB Application Deployment Web Example Guide*”.

Work 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 8-14.

Work 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);
```

.NET Remoting

- “What Is .NET Remoting?” on page 9-2
- “Your Role in Building Distributed Applications” on page 9-4
- “.NET Remoting Prerequisites” on page 9-5
- “Select the Best Method of Accessing Your Component: MWArray API or Native .NET API” on page 9-6
- “Creating a Remotable .NET Component” on page 9-8
- “Enabling Access to a Remotable .NET Component” on page 9-11

What Is .NET Remoting?

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 without restrictions.

What’s the Difference Between WCF and .NET Remoting?

You generate native .NET objects using both .NET Remoting (“Creating a Remotable .NET Component”) and native .NET types(WCF).

What's the difference between these two technologies and which should you use?

WCF is an end-to-end *Web Service*. Many of the advantages afforded by .NET Remoting—a wide selection of protocol interoperability, for instance—can be achieved with a WCF interface, in addition to having access to a richer, more flexible set of native data types. .NET Remoting can only support native objects.

WCF offers more robust choices in most every aspect of Web-based development, even implementation of a Java client, for example.

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.	<ul style="list-style-type: none"> Writes and deploys MATLAB code. Creates a deployable, remotable .NET component as in “Creating a Remotable .NET Component” on page 9-8.
.NET programmer	Exposes .NET applications to end users.	<ul style="list-style-type: none"> Writes client/server code to access the remotable component as in “Using the MWArray API” on page 9-11 or “Using the Native .NET API: Magic Square” on page 9-16.

.NET Remoting Prerequisites

Before you enable .NET Remoting for your deployable component, be aware of the following:

- You cannot enable both .NET Remoting and Windows Communication Foundation (WCF) (as described in “Create Windows Communications Foundation (WCF)TM-Based Components” on page 7-12).
- It is important to determine if you derive more benefit and cost savings by using the MArray API or the native .NET API. Evaluate if .NET Remoting is appropriate for your deployable component by reading “Select the Best Method of Accessing Your Component: MArray API or Native .NET API” on page 9-6.

Select 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 marshaling and formatting services for all basic MATLAB data types including sparse arrays, structures, and cell arrays. This API requires the MATLAB runtime 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 9-11.

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 remotable component using the native interface API, as it does not require the client machine to have the MATLAB runtime 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: Magic Square" on page 9-16.

Features of the MWArray API Compared With the Native .NET API

	MWArray API	Native .NET API
Marshaling/formatting for all basic MATLAB types	X	
Pass arguments and return values using standard .NET types		X
Access to remotable component from client without installed MATLAB		X
Access to remotable component from client		X

	MWArray API	Native .NET API
without installed MATLAB runtime (see “Using the Native .NET API: Cell and Struct” on page 9-22).		

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: Cell and Struct” on page 9-22 for details.

Creating a Remotable .NET Component

In this section...

“Building a Remotable Component Using the Library Compiler App” on page 9-8

“Building a Remotable Component Using the mcc Command” on page 9-9

“Files Generated by the Compilation Process” on page 9-10

Building a Remotable Component Using the Library Compiler App

- 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\VSVersion\NET\MagicRemoteExample\MWArray
```

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\VSVersion\NET\MagicRemoteExample\Native
```

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:

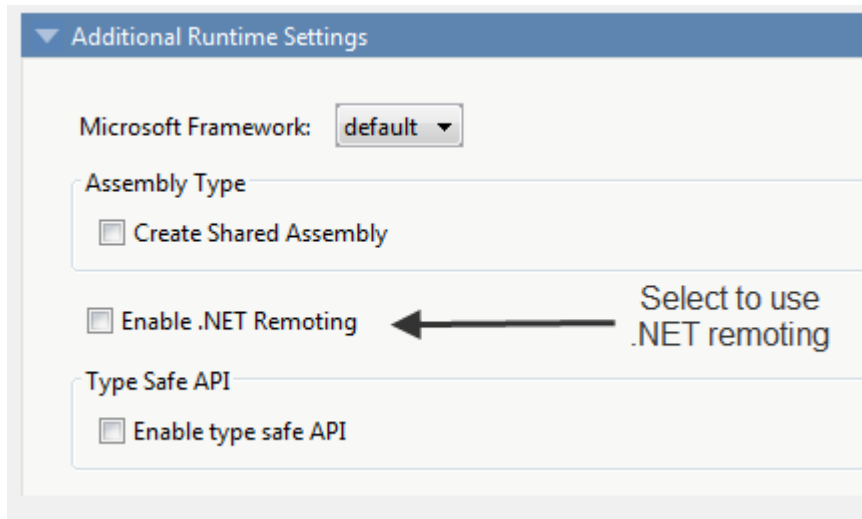
```
function y = makesquare(x)

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

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

- 3 Click the **Library Compiler** app in the apps gallery.
- 4 In the Additional Runtime Settings area, select **Enable .NET Remoting**.



- 5 Build the .NET component. See the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11 for more details.

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`.

To build a private remotable component, the `mcc` command to build the component for the .NET 2.0 Framework will look similar to:

```
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 `for_redistribution_files_only` 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 a MATLAB runtime on the client when using this interface.

Enabling Access to a Remotable .NET Component

In this section...

“Using the MWArray API” on page 9-11

“Using the Native .NET API: Magic Square” on page 9-16

“Using the Native .NET API: Cell and Struct” on page 9-22

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 “Select the Best Method of Accessing Your Component: MWArray API or Native .NET API” on page 9-6.

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: Magic Square” on page 9-16.

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 9-8. You can also perform these steps using the native .NET API as discussed in “Using the Native .NET API: Magic Square” on page 9-16.

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\for_redistribution_files_only\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 9-11.

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\for_redistribution_files_only\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}",
```


- 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 MATLAB runtime 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: Magic Square

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 “Select the Best Method of Accessing Your Component: MWArray API or Native .NET API” on page 9-6.

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 9-11.

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 9-8.

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\for_redistribution_files_only\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

The C# code for the server is in 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 9-17. 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\for_redistribution_files_only\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

The C# code for the client is 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 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 MATLAB runtime 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: Cell and Struct

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 a MATLAB runtime 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 a MATLAB runtime.

The `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\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 Library Compiler App” on page 9-8 or “Building a Remotable Component Using the `mcc` Command” on page 9-9.

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\for_redistribution_files_only\component_nameNative.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 9-23. 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\for_redistribution_files_only\Icomponent_nameNative`. 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

The C# 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 window and cd to NativeStructCellClient\bin\x86\v2.0\Debug.
- 2 Run NativeStructCellClient.exe. After the MATLAB runtime 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 client and add more 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 MATLAB runtime 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();
        }
    }
}

```


Troubleshooting

This chapter provides some solutions to common problems encountered using the MATLAB Builder NE product.

- “Viewing the Latest Build Log” on page 10-2
- “Failure to Find a Required File” on page 10-3
- “Diagnostic Messages” on page 10-4

Viewing the Latest Build Log

To view the log of your most recent build process, open `/projName/PackagingLog.txt`.

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 MATLAB runtime 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 MATLAB runtime 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 MATLAB runtime folder) and *ver* represents the 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 may 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.	<p>On the target machine where the COM component is to be used:</p> <ol style="list-style-type: none"> 1 Use the <code>extractCTF.exe</code> utility to decompress the <code>.ctf</code> file generated by the builder when you built the COM component. 2 Look at the files in the deployable archive, and note the path for the DLL. 3 Add this path to the system path. <p>See the MATLAB Compiler documentation for</p>

Message	Probable Cause	Suggested Solution
		more information about <code>extractctf.exe</code> .
	You may get this error if you do not have appropriate permissions to deploy a COM component on a particular system.	See “Add-in and COM Component Registration”.
Error in <i>component_name.class_name.x</i> : Error getting data conversion flags.	This is often caused by <code>mwcomutil.dll</code> not being registered.	<ol style="list-style-type: none"> 1 Open a DOS window. 2 Change folders to <code>matlabroot\runtime\architecture</code>. 3 Run the following command: <code>mwregsvr mwcomutil.dll</code> <p>(<i>matlabroot</i> is your root MATLAB folder.)</p>
Error in VBAProject: ActiveX component can't create object.	<ul style="list-style-type: none"> • Project DLL is not registered. • An incompatible MATLAB DLL exists somewhere on the system path. 	<p>If the DLL is not registered,</p> <ol style="list-style-type: none"> 1 Open a DOS window. 2 Change folders to <code>projectdir\distrib</code>. 3 Run the following command: <code>mwregsvr projectdll.dll</code> <p>(<i>projectdir</i> represents the location of your project files).</p>
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 <code>new</code>). See “Classes and Methods” on page 5-3 in 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.

Message	Probable Cause	Suggested Solution
<p>QueryInterface for interface <COM OBJECT NAME> failed.</p>	<p>You might be using the incorrect number and/or type of function parameters to call into your COM object.</p>	<p>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:</p> <ul style="list-style-type: none"> • Use a Variant data type for the return type of the COM object. • Use doubles as default numeric input parameters (rather than integers). <p>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.</p>
<p>Showing a modal dialog box or form when the application is not running in UserInteractive mode is not a valid operation. Specify the ServiceNotification or DefaultDesktopOnly style to display a notification from a service application.</p>	<p>This warning occurs when ASP.NET code tries to bring up a dialog box.</p> <p>If occurs because <code>getframe()</code> makes the figure window visible before performing the capture and thus fails when running in IIS. <code>msgbox()</code> calls in MATLAB code cause the warning to appear also.</p>	<p>Work around this problem by doing the following:</p> <ol style="list-style-type: none"> 1 Open the Windows Control Panel. 2 Open Services. 3 From the list of services, select and open the IIS Admin service. 4 In the Properties dialog, on the Log On tab, select Local System Account. 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\
CallmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name
dmakeerror_error2,line at 14.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CallmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name
dmakeerror_error1,line at 11.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CallmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name dmakeerror,line at 4.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CallmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\callmakeerror.m,name
callmakeerror,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 CallldmakeerrComp.Callldmakeerr.callldmakeerror() in
h:\compiler\g388611\cathy\MagicDemoComp\src\
Callldmakeerr.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 11-3
- “Overview of Data Conversion Classes” on page 11-10
- “MWArray Class Specification” on page 11-16
- “Deployment Product Terms” on page 11-17

Requirements for the MATLAB Builder NE Product

In this section...
“System and Compiler Requirements” on page 11-2
“Path Modifications Required for Accessibility” on page 11-5

System and 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 system and compiler software supported by MATLAB, the builders, 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.

Data Conversion Rules

In this section...

“Managed Types to MATLAB Arrays” on page 11-3

“MATLAB Arrays to Managed Types” on page 11-3

“.NET Types to MATLAB Types” on page 11-4

“Character and String Conversion” on page 11-9

“Unsupported MATLAB Array Types” on page 11-9

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

Native .NET Type	MATLAB Array	Comments
System.Double	double	—
System.Single	single	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 double type.
System.Int64	int64	
System.Int32	int32	
System.Int16	int16	
System.Byte	int8	
System.String	char	None
System.Boolean	logical	None

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

.NET Types to MATLAB Types

In order to create .NET interfaces that describe the type-safe API of a MATLAB Builder NE generated component, you must decide on the .NET types used for input and output parameters.

When choosing input types, consider how .NET inputs become MATLAB types. When choosing output types, consider the inverse conversion

The following tables list the data conversion results and rules used to convert .NET types to MATLAB arrays and MATLAB arrays to .NET types.

Note: Invalid conversions result in a thrown `ArgumentException`

Conversion Results: .NET Types to MATLAB Types

.NET Type	Converts to MATLAB Type
NumericType <ul style="list-style-type: none"> • System.Double • System.Single • System.Byte • System.Int16 • System.Int32 • System.Int64 • System.Int64 	numeric
System.Boolean	logical
System.Char	char
System.String	
NumericType[N]	NumericType[1,N]
NumericType[P _n ,...,P ₁ ,M,N]	NumericType[M,N,P ₁ ,...,P _n]
System.Boolean[N]	logical [1,N]
System.Boolean[P _n ,...,P ₁ ,M,N]	logical [M,N,P ₁ ,...,P _n]
System.Char[N]	char [1,N]
System.Char[P _n ,...,P ₁ ,M,N]	char [M,N,P ₁ ,...,P _n]
System.String[N]	char [N,max_string_length]
System.String[P _n ,...,P ₁ ,N]	char [N,max_string_length, P ₁ ,...,P _n]
Scalar .NET struct	MATLAB struct constructed from public instance fields of the .NET struct
.NET struct [N]	MATLAB struct [1,N] where each element is constructed from public instance fields of the .NET struct
.NET struct [M,N]	MATLAB struct [M,N] where each element is constructed from public instance fields of the .NET struct

.NET Type	Converts to MATLAB Type
native.MWStructArray	struct
native.MWCellArray	cell
Hashtable	struct
Dictionary <K,V> Where K = string and V = scalar or array of [Numeric, boolean, Char, String]	struct
ArrayList	cell
Any other .NET type in the default application domain	.NET object
Any other serializable .NET type in a non-default application domain	.NET object

Conversion Rules: MATLAB Numeric Types to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
numeric	Scalar	The type must be scalar in MATLAB. For example, a 1 X 1 int in MATLAB.
	Vector	The type must be a vector in MATLAB. For example, a 1 X N or N X 1 int array in MATLAB.
	N-dimensional array	The N-dimensional array type specified by the user must match the rank of the MATLAB numeric array.

Tip When converting MATLAB numeric arrays, widening conversions are allowed. For example, an int can be converted to a double. The type specified must be a numeric type that is equal or wider. Narrowing conversions throw an ArgumentException.

Caution .NET types are not as flexible as MATLAB types. Take care and test appropriately with .NET outputs before integrating data into your applications.

Conversion Rules: MATLAB Char Arrays to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
char	Char	The char must be scalar.
	Char array	The N -dimensional Char type must match the rank of the MATLAB char array.
	String	MATLAB char array must be $[1, N]$
	String array	The N -dimensional MATLAB char array can be converted to $(N-1)$ -dimensional array of type String.

Conversion Rules: MATLAB Logical Arrays to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
logical	Boolean	The logical must be scalar.
	Boolean[]	The MATLAB logical array must be $[1, N]$ or $[N, 1]$.
	Boolean array	The N -dimensional Boolean array must match the rank of the MATLAB logical array.

Conversion Rules: Cell Array to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
cell	System.Array	The N -dimensional MATLAB cell array is converted to an N -dimensional System.Array of type object.

To Convert This MATLAB Type:	To this:	Follow these rules:
	ArrayList	The MATLAB cell array must be a vector.

Caution If the MATLAB cell array contains a struct, it is left unchanged. All other types are converted to native types. Any nested cell array is converted to a System.Array matching the dimension of the cell array, as illustrated in this code snippet:

```
Let C = {[1,2,3], {[1,2,3]}, 'Hello world'}
% be a cell
C can be converted to an object[1,3] where
object[1,1] contains int[, ]
object[1,2] contains an object[1,1]
whose first element in an
int[, ] object[1,3] contains char[, ].
```

Note: Any nested cell array is converted to a System.Array that matches the dimension of the cell array

Conversion Rules: Struct to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
struct	.NET struct	The name and number of public fields in the specified .NET struct must match the name and number of fields in the MATLAB struct.
	Hashtable	A scalar struct can be converted to a Hashtable. Any nested struct will also be converted to a Hashtable. If the nested struct is not a scalar, then an ArgumentException is

To Convert This MATLAB Type:	To this:	Follow these rules:
		thrown. The dictionary key must be of type <code>String</code> .

Conversion Rules: .NET Objects in MATLAB to .NET Native Objects

To Convert this MATLAB Type:	To this:	Follow these rules:
.NET object	Type or super-type of the containing object	A .NET object in MATLAB can only be converted to a type or a super-type.

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 11-10
“Returning Data from MATLAB to Managed Code” on page 11-11
“Example of MWNumericArray in a .NET Application” on page 11-11
“Interfaces Generated by the MATLAB Builder NE Product” on page 11-11

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.

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 `MWArrays`.
- 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	<code>function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)</code>
API if there are no input arguments	<code>public MWArray foo()</code>
API if there are one or more input arguments	<code>public MWArray foo(MWArray In1, MWArray In2 ... MWArray inN)</code>
API if there are optional input arguments	<code>public MWArray foo(MWArray In1, MWArray In2, ... MWArray inN params MWArray[] varargin)</code>

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	<code>function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)</code>
API if there are no input arguments	<code>public MWArray[] foo(int numArgsOut)</code>
API if there is one input argument	<code>public MWArray [] foo(int numArgsOut, MWArray In1)</code>
API if there are two to <i>N</i> input arguments	<code>public MWArray[] foo(int numArgsOut, MWArray In1, MWArray In2, \... MWArray InN)</code>
API if there are optional arguments, represented by the <code>varargin</code> argument	<code>public MWArray[] foo(int numArgsOut, MWArray in1, MWArray in2, MWArray InN, params MWArray[] varargin)</code>

Details about the arguments for these samples of standard signatures are shown in the following table.

Argument	Description	Details
<i>numArgsOut</i>	Number of outputs	<p>An integer indicating the number of outputs you want the method to return.</p> <p>The value of <i>numArgsOut</i> must be less than or equal to the MATLAB function <i>nargout</i>.</p> <p>The <i>numArgsOut</i> argument must always be the first argument in the list.</p>
<i>In1</i> , <i>In2</i> , ... <i>InN</i>	Required input arguments	<p>All arguments that follow <i>numArgsOut</i> in the argument list are inputs to the method being called.</p> <p>Specify all required inputs first. Each required input must be of type <i>MWArray</i> or one of its derived types.</p>
<i>varargin</i>	Optional inputs	<p>You can also specify optional inputs if your MATLAB code uses the <i>varargin</i> input: list the optional inputs, or put them in an <i>MWArray[]</i> argument, placing the array last in the argument list.</p>
<i>Out1</i> , <i>Out2</i> , ... <i>OutN</i>	Output arguments	<p>With the standard calling interface, all output arguments are returned as an array of <i>MWArrays</i>.</p>

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 MArray [] ArgsOut,
     MArray[] ArgsIn)
```

where the arguments are as follows:

<code>numArgsOut</code>	Number of outputs	<p>Same as standard interface.</p> <p>An integer indicating the number of outputs you want to return.</p> <p>This number generally matches the number of output arguments that follow. The <code>varargout</code> array counts as just one argument, if present.</p>
<code>ref MArray [] ArgsOut</code>	Output arguments	<p>Following <code>numArgsOut</code> 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.</p> <p>A <code>ref</code> attribute prefaces all output arguments indicating that these arrays are passed by reference.</p>
<code>MArray[] ArgsIn</code>	Input arguments	<p><code>MArray</code> types or a supported .NET primitive type.</p> <p>When you pass an instance of an <code>MArray</code> type, the underlying MATLAB array is passed directly to the called function. Native types are first converted to <code>MArray</code> types.</p>

MWArray Class Specification

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

Deployment Product Terms

A

Add-in — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic.

Application program interface (API) — A set of classes, methods, and interfaces that is used to develop software applications. Typically an API is used to provide access to specific functionality. See *MWArray*.

Application — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET.

B

Binary — See *Executable*.

Boxed Types — Data types used to wrap opaque C structures.

Build — See *Compile*.

C

Class — A user-defined type used in C++, C#, and Java, among other object-oriented languages, that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes relate in a class hierarchy. One class is a specialization (a *subclass*) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a *MATLAB class*

Compile — In MATLAB Compiler terminology, to compile MATLAB code involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code is compiled into a Java

package, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Builder EX, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Builder NE, an executable component, to be integrated with Microsoft COM applications.

Console application — Any application that is executed from a system command prompt window.

D

Data Marshaling — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the `MWArray` API—must be performed manually, often at great cost.

Deploy — The act of integrating MATLAB code into a larger-scale computing environment, usually to an enterprise application, and often to end users.

Deployable archive — The deployable archive is embedded by default in each generated binary by MATLAB Compiler. It houses the deployable package. All MATLAB-based content in the deployable archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details” in the MATLAB Compiler documentation.

DLL — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

E

Empties — Arrays of zero (0) dimensions.

Executable — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

F

Fields — For this definition in the context of MATLAB Data Structures, see *Structs*.

Fields and Properties — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

I

Integration — Combining deployed MATLAB code's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment.

Instance — For the definition of this term in context of MATLAB Production Server software, see *MATLAB Production Server Server Instance*.

J

JAR — Java archive. In computing software, a JAR file (or Java ARchive) aggregates many files into one. Software developers use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

Java-MATLAB Interface — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

JDK — The *Java Development Kit* is a free Oracle® product which provides the environment required for programming in Java.

JMI Interface — see *Java-MATLAB Interface*.

JRE — *Java Run-Time Environment* is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

M

Magic Square — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

MATLAB Runtime — An execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB.

MATLAB Runtime Singleton — See *Shared MATLAB Runtime Instance*.

MATLAB Runtime Workers — A MATLAB runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MATLAB runtime session, using the `--num-workers` options in the server configurations file.

MATLAB Production Server Client — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

MATLAB Production Server Configuration — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, “`main_config`”.

MATLAB Production Server Server Instance — A logical server configuration created using the `mps -new` command in MATLAB Production Server software.

MATLAB Production Server Software — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, Web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

Marshaling — See *Data Marshaling*.

mbuild — MATLAB Compiler command that compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

mcc — The MATLAB command that invokes MATLAB Compiler. It is the command-line equivalent of using the compiler apps.

Method Attribute — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

mxArray interface — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

MWArray interface — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`. There are different implementations of the `MWArray` proxy for each application programming language.

P

Package — The act of bundling the deployed MATLAB code, along with the MATLAB runtime and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of loose artifacts that can be used for testing or building a custom installer.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a *pool*, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept in for Microsoft Windows.

Shared MATLAB Runtime Instance — When using MATLAB Builder NE or MATLAB Builder JA, you can create a shared MATLAB runtime instance, also known as a *singleton*. For builder NE, this only applies to COM components. When you invoke MATLAB Compiler with the `-S` option through the builders (using either `mcc` or a compiler app), a single MATLAB runtime instance is created for each COM component or Java package in an application. You reuse this instance by sharing it among all subsequent class instances. Such sharing results in more efficient memory usage and eliminates the MATLAB runtime startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the deployed MATLAB files. MATLAB Builder NE and MATLAB Builder EX are designed to create singletons by default for .NET assemblies and COM components, respectively. For more information, see “Sharing a MATLAB Runtime Instance in COM or Java Applications”.

State — The present condition of a the MATLAB, or MATLAB runtime, runtime. MATLAB functions often carry state in the form of variable values. The MATLAB workspace itself also maintains information about global variables and path settings. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions.

Structs — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

System Compiler — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio.

T

Thread — A portion of a program that can run independently of and concurrently with other portions of the program. See *pool* for additional information on managing the number of processing threads available to a server instance.

Type-safe interface — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application.

W

Web Application Archive (WAR) —In computing, a Web Application Archive is a JAR file used to distribute a collection of `JavaServer` pages, servlets, Java classes, XML files, tag libraries, and static Web pages (HTML and related files) that together constitute a Web application.

Webfigure — A MathWorks representation of a MATLAB figure, rendered on the Web. Using the WebFigures feature, you display MATLAB figures on a Web site for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the Web, without the need to download MATLAB or other tools that can consume costly resources.

Windows Communication Foundation (WCF) — The Windows Communication Foundation™ is an application programming interface in the .NET Framework for building connected, service-oriented, Web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed by client applications.

Function Reference

componentinfo
deploytool
enableTSUtilsfromNetworkDrive
figToImStream
libraryCompiler
mcc
ntswrap

componentinfo

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

<i>component_name</i>	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.
<i>major_revision_number</i>	Component major revision number. If the argument is not supplied, information is returned on all major revisions.
<i>minor_revision_number</i>	Component minor revision number. Default value is 0.

Description

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

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

`info = componentinfo(component_name, major_revision_number)` returns information for the most recent minor revision corresponding to *major_revision_number* of *component_name*.

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

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

When you supply a component name, *major_revision_number* and *minor_revision_number* are interpreted as shown next.

Value	Information Returned
> 0	Information on a specific major and minor revision.
0	Information on the most recent revision. When omitted, <i>minor_revision_number</i> is assumed to be 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.
Interfaces	An array of structures defining all interface definitions in the type library. Each structure contains two fields: <ul style="list-style-type: none"> • Name - Interface name. • IID - Interface GUID.
CoClasses	An array of structures defining all COM classes in the component. Each structure contains these fields:

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

Examples

Function Call	Returned Information
Info = componentinfo	Information for all installed components.
Info = componentinfo('mycomponent')	Information for all revisions of mycomponent.

Function Call	Returned Information
Info = componentinfo('mycomponent',1,0)	Information for revision 1.0 of mycomponent.

More About

Tips

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.

deploytool

Compile and package functions for external deployment

Syntax

```
deploytool  
deploytool project_name  
deploytool -build project_name  
deploytool -package project_name
```

Description

`deploytool` opens the MATLAB Compiler app.

`deploytool project_name` opens the MATLAB Compiler app with the project preloaded.

`deploytool -build project_name` runs the MATLAB Compiler to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the MATLAB Compiler to build and package the specified project. The installer is generated.

Examples

Create a New Compiler Project

Open the compiler to create a new project.

```
deploytool
```

Package an Application using an Existing Project

Open the compiler to build a new application using an existing project.

```
deploytool -package my_magic
```

Input Arguments

project_name — name of the project to be compiled

string

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

enableTSUtilsfromNetworkDrive

Enable access to .NET commands from network drive

Syntax

```
enableTSUtilsfromNetworkDrive
```

Description

`enableTSUtilsfromNetworkDrive` adds an entry for the MATLAB Builder NE module to the security policy on your machine.

Tips

- Administrator privileges are required to run this command.

Examples

To enable use of MATLAB Builder NE on a system, enter the following on the MATLAB command line after logging in with Administrator privileges:

```
enableTSUtilsfromNetworkDrive
```

figToImStream

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' );
```


libraryCompiler

Build and package functions for use in external applications

Syntax

```
libraryCompiler  
libraryCompiler project_name  
libraryCompiler -build project_name  
libraryCompiler -package project_name
```

Description

`libraryCompiler` opens the MATLAB shared library compiler for the creation of a new compiler project

`libraryCompiler project_name` opens the MATLAB shared library compiler app with the project preloaded.

`libraryCompiler -build project_name` runs the MATLAB shared library compiler to build the specified project. The installer is not generated.

`libraryCompiler -package project_name` runs the MATLAB shared library compiler to build and package the specified project. The installer is generated.

Examples

Create a New Project

Open the library compiler to create a new project.

```
libraryCompiler
```

Package a Function using an Existing Project

Open the library compiler using an existing project.

```
libraryCompiler -package my_magic
```

Input Arguments

project_name — name of the project to be compiled
string

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

mcc

Compile MATLAB functions for deployment

Syntax

```

mcc -e | -m [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-v] [-w option [:msg]] [-Y filename] mfilename
mcc -l [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -c [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -W cpplib:library_name -T link:lib [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-g] [-I directory...] [-K] [-M string] [-N] [-o filename] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -W dotnet:assembly_name, [className], [framework_version], security, remote_type -T link:lib [-a filename...] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]
mcc -W excel:addin_name, [className], [version] -T link:lib [-a filename...] [-b] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-u] [-v] [-w option [:msg]] [-Y filename] mfilename...
mcc -W 'java:packageName, [className]' [-a filename...] [-b] [-B filename [:arg...]] [-C] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]
mcc -W CTF:archive_name [-a filename...] [-b] [-B filename [:arg...]] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]
mcc -W mpsxl:archive_name, [className], [version] -T link:lib [-replaceBlankWithNaN] [-convertDateToString] [-replaceNaNWithZero] [-convertNumericToDate] [-a filename...] [-b] [-B filename [:arg...]] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] mfilename... [class{className: [mfilename...]}...]

```

```
mcc -H -W mpsxl:archive_name, [className], [version] -T link:lib [-
replaceBlankWithNaN] [-convertDateToString] [-replaceNaNWithZero] [-
convertNumericToDate] [-a filename...] [-b] [-B filename [:arg...]] [-d outFolder] [-f
filename] [-I directory...] [-K] [-M string] [-N] [-p path...] [-R option] [-S] [-v] [-w
option [:msg]] [-Y filename] filename... [class{className: [mfilename...]}...]
mcc -H -W hadoop:archive_name,CONFIG:configFile [-a filename...] [-b] [-B
filename [:arg...]] [-d outFolder] [-f filename] [-I directory...] [-K] [-M string]
[-N] [-p path...] [-R option] [-S] [-v] [-w option [:msg]] [-Y filename] filename...
[class{className: [mfilename...]}...]
mcc -?
```

Description

`mcc -m mfilename` compiles the function into a standalone application.

This is equivalent to `-W main -T link:exe`.

`mcc -e mfilename` compiles the function into a standalone application that does not open an MS-DOS[®] command window.

This is equivalent to `-W WinMain -T link:exe`.

`mcc -l mfilename...` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This is equivalent to `-W lib:libname -T link:lib`.

`mcc -c mfilename...` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:libname -T codegen`.

`mcc -W cpplib:library_name -T link:lib mfilename...` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

`mcc -W dotnet:assembly_name,className,framework_version,security,remote_type -T link:lib mfilename...` creates a .NET assembly from the specified files.

- *assembly_name* — Specifies the name of the assembly and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *className* — Specifies the name of the .NET class to be created.
- *framework_version* — Specifies the version of the Microsoft .NET Framework you want to use to compile the assembly. Specify either:
 - *0.0* — Use the latest supported version on the target machine.
 - *version_major.version_minor* — Use a specific version of the framework.

Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.

- *security* — Specifies whether the assembly to be created is a private assembly or a shared assembly.
 - To create a private assembly, specify **Private**.
 - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *remote_type* — Specifies the remoting type of the assembly. Values are **remote** and **local**.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}...className` specifies the name of the class to create using *mfilename*.

`mcc -W excel:addin_name,className,version -T link:lib mfilename...` creates a Microsoft Excel add-in from the specified files.

- *addin_name* — Specifies the name of the addin and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *addin_name* as the default.
- *version* — Specifies the version of the add-in specified as *major.minor*.
 - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
 - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

`mcc -W 'java:packageName,className' mfilename...` creates a Java package from the specified files.

- *packageName* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *packageName*.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}...className` specifies the name of the class to create using *mfilename*.

`mcc -W CTF:archive_name` instructs the compiler to create a deployable archive that is deployable in a MATLAB Production Server instance.

`mcc -W mpsxl:addin_name,className,version input_marshall_options output_marshall_options -T link:lib mfilename...` creates an MATLAB Production Server integrated Microsoft Excel add-in from the specified files.

- *addin_name* — Specifies the name of the add-in and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *addin_name* as the default.
- *version* — Specifies the version of the add-in specified as *major.minor*.
 - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
 - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.
- *input_marshall_options* — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
 - `-replaceBlankWithNaN` — Specifies that blanks in Microsoft Excel are marshaled into MATLAB NaNs. If you do not specify this flag, blanks are marshaled as 0.
 - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB strings. If you do not specify this flag, dates are marshaled as MATLAB doubles.
- *output_marshall_options* — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.

- `-replaceNaNWithZero` — Specifies that MATLAB NaNs are marshaled into Microsoft Excel 0s . If you do not specify this flag, NaNs are marshaled as Visual Basic #QNANs.
- `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

`mcc -H -W hadoop:archiveName,CONFIG:configFile` generates deployable archive that can be run as a job by Hadoop®.

- *archiveName* — Specifies the name of the generated archive.
- *configFile* — Specifies the path to the Hadoop settings file. See “Hadoop Settings File”.

`mcc -?` displays help.

Tip You can issue the `mcc` command either from the MATLAB command prompt or the DOS or UNIX command line.

Options

-a Add to Archive

Add a file to the deployable archive using

```
-a filename
```

to specify a file to be directly added to the deployable archive. Multiple `-a` options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If only a folder name is included with the `-a` option, the entire contents of that folder are added recursively to the deployable archive. For example:

```
mcc -m hello.m -a ./testdir
```

In this example, `testdir` is a folder in the current working folder. All files in `testdir`, as well as all files in subfolders of `testdir`, are added to the deployable archive, and the folder subtree in `testdir` is preserved in the deployable archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the deployable archive and subfolders of the given path are not processed recursively. For example:

```
mcc -m hello.m -a ./testdir/*
```

In this example, all files in `./testdir` are added to the deployable archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

In this example, all files with the extension `.m` under `./testdir` are added to the deployable archive and subfolders of `./testdir` are not processed recursively.

All files added to the deployable archive using `-a` (including those that match a wildcard pattern or appear under a folder specified using `-a`) that do not appear on the MATLAB path at the time of compilation causes a path entry to be added to the deployed application's run-time path so that they appear on the path when the deployed code executes.

When files are included, the absolute path for the DLL and header files is changed. The files are placed in the `.\exe_mcr\` folder when the deployable archive is expanded. The file is not placed in the local folder. This folder is created from the deployable archive the first time the application is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the `exe_mcr\exe` folder.

Caution If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

Note: Currently, `*` is the only supported wildcard.

Note: If the `-a` flag is used to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

-b Generate Excel Compatible Formula Function

Generate a Visual Basic file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function. This option requires MATLAB Builder EX.

-B Specify Bundle File

Replace the file on the `mcc` command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle file `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See “Using Bundle Files to Build MATLAB Code” for a list of the bundle files included with MATLAB Compiler.

-C Do Not Embed Deployable Archive by Default

Override automatically embedding the deployable archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.

-d Specified Folder for Output

Place output in a specified folder. Use

```
-d outFolder
```

to direct the generated files to *outFolder*.

-e Suppress MS-DOS Command Window

Suppress appearance of the MS-DOS command window when generating a standalone application. Use `-e` in place of the `-m` option. This option is available for Windows only. Use with `-R` option to generate error logging as such:

```
mcc -e -R -logfile -R 'filename' -v function_name
```

or:

```
mcc -e -R '-logfile,logfile' -v function_name
```

For example, to build a standalone from function `foo.m` that suppresses the MS-DOS command window, and specifying error logging to a text file, enter this command:

```
mcc -e -R '-logfile,errorlog.txt' -v foo.m
```

-f Specified Options File

Override the default options file with the specified options file. Use

```
-f filename
```

to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to `mbuild`.

-g Generate Debugging Information

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MATLAB runtime, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

-G Debug Only

Same as `-g`.

-I Add Folder to Include Path

Add a new folder path to the list of included folders. Each `-I` option adds a folder to the beginning of the list of paths to search. For example,

```
-I <directory1> -I <directory2>
```

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

-K Preserve Partial Output Files

Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

-m Generate a Standalone Application

Macro to produce a standalone application. This macro is equivalent to the defunct:

```
-W main -T link:exe
```

Use the `-e` option instead of the `-m` option to generate a standalone application while suppressing the appearance of the MS-DOS command window.

Note: Using the `-e` option requires the application to successfully compile with a Microsoft compiler.

-M Direct Pass Through

Define compile-time options. Use

```
-M string
```

to pass `string` directly to `mbuild`. This provides a useful mechanism for defining compile-time options, e.g., `-M "-Dmacro=value"`.

Note: Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

-N Clear Path

Passing `-N` effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- `matlabroot\toolbox\matlab`
- `matlabroot\toolbox\local`
- `matlabroot\toolbox\compiler\deploy`

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under `matlabroot\toolbox`.

-o Specify Output Name

Specify the name of the final executable (standalone applications only). Use

```
-o outputfile
```

to name the final executable output of MATLAB Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows standalone applications).

-p Add Folder to Path

Use in conjunction with the required option `-N` to add specific folders (and subfolders) under `matlabroot\toolbox` to the compilation MATLAB path in an order sensitive way. Use the syntax

```
-N -p directory
```

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included follow.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.

- If a folder is included with `-p` that is not on the original MATLAB path, that folder is not included in the compilation. (You can use `-I` to add it.)

If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the folder is added to the head of the path, as it normally would be with `-I`.

-R Run-Time

Provides MATLAB runtime run-time options. The syntax is as follows:

`-R option`

Option	Description
<code>-logfile, filename</code>	Specify a log file name.
<code>-nodisplay</code>	Suppress the MATLAB <code>nodisplay</code> run-time warning.
<code>-nojvm</code>	Do not use the Java Virtual Machine (JVM).
<code>-startmsg</code>	Customizable user message displayed at initialization time.
<code>-completemsg</code>	Customizable user message displayed when initialization is complete.

Note: Not all `-R` options are available for all `mcc` targets.

Caution When running on Mac OS X, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

-S Create Singleton MATLAB Runtime Context

The standard behavior for the MATLAB runtime is that every instance of a class gets its own runtime context. This runtime context includes a global MATLAB workspace for variables such as the path and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This

ensures that changes made to the global, or base, workspace in one instance of the class does not effect other instances of the same class.

In a singleton MATLAB runtime, all instances of a class share the runtime context. If multiple instances of a class are created, they use the runtime context created by the first instance. This saves start up time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all of the class instances. For example, if `instance1` creates a global variable `A` in a singleton MATLAB runtime, the `instance2` will be able to use variable `A`.

-T Specify Target Stage

Specify the output target phase and type.

Use the syntax `-T target` to define the output type. Target values are as follow.

Target	Description
<code>compile:exe</code>	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a standalone application.
<code>compile:lib</code>	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a shared library/DLL.
<code>link:exe</code>	Same as <code>compile:exe</code> plus links object files into a standalone application.
<code>link:lib</code>	Same as <code>compile:lib</code> plus links object files into a shared library/DLL.

-u Register COM Component for the Current User

Register COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

-v Verbose

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

-w Warning Messages

Display warning messages. Use the syntax

`-w option [:<msg>]`

to control the display of warnings. This table lists the syntaxes.

Syntax	Description
<code>-w list</code>	Generate a table that maps <code><string></code> to warning message for use with <code>enable</code> , <code>disable</code> , and <code>error</code> . “Warning Messages”, lists the same information.
<code>-w enable</code>	Enable complete warnings.
<code>-w disable[:<string>]</code>	Disable specific warnings associated with <code><string></code> . “Warning Messages”, lists the <code><string></code> values. Omit the optional <code><string></code> to apply the <code>disable</code> action to all warnings.
<code>-w enable[:<string>]</code>	Enable specific warnings associated with <code><string></code> . “Warning Messages”, lists the <code><string></code> values. Omit the optional <code><string></code> to apply the <code>enable</code> action to all warnings.
<code>-w error[:<string>]</code>	Treat specific warnings associated with <code><string></code> as an error. Omit the optional <code><string></code> to apply the <code>error</code> action to all warnings.
<code>-w off[:<string>] [<filename>]</code>	Turn warnings off for specific error messages defined by <code><string></code> . You can also narrow scope by specifying warnings be turned off when generated by specific <code><filename></code> s.

Syntax	Description
<code>-w on[:<string>] [<filename>]</code>	Turn warnings on for specific error messages defined by <i><string></i> . You can also narrow scope by specifying warnings be turned on when generated by specific <i><filename></i> s.

It is also possible to turn warnings on or off in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using `isdeployed`) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

-Y License File

Use

```
-Y license.lic
```

to override the default license file with the specified argument.

ntswrap

Generates type-safe API

Syntax

```
ntswrap.exe [-c namespace.class] [-i interface_name]  
[-a assembly_name]
```

Description

Available as a MATLAB function or Windows console executable.

```
ntswrap.exe [-c namespace.class] [-i interface_name]  
[-a assembly_name] accepts command line switches in any order.
```

Run `ntswrap` for “Build Your Component and Generate Your Type-Safe API” on page 7-8 with a MATLAB Builder NE generated assembly.

Arguments

Inputs

-a *.NET_native_interface.dll*

Absolute or relative path to assembly containing .NET statically-typed interface, referenced by -i switch.

-b *MATLAB_Builder_NE_Component.dll*

Path to folder containing MATLAB Builder NE component that defines component referenced by -c switch

-c *component_class_name*

Namespace-qualified name of MATLAB Builder NE component in assembly identified by path in -b switch

-d

Enables debugging of the type-safe API assembly

Incompatible with `-s`.

`-i interface_name`

Namespace-qualified name of user-supplied interface in assembly identified by path in `-a` switch

`-k`

Keep generated type safe API source code; do not delete after processing

`-n namespace_containing_generated_type-safe_API_class`

Optional. If specified, places generated type-safe API in specified namespace

`-o output_folder`

Optional. If specified, all output files will be written to specified, preallocated folder

`-s`

Generate source code only; do not compile type-safe API source into an assembly

`-v vx.x`

Version of Microsoft .NET Framework (CSC compiler) used to generate type-safe API assembly (for example `v2.0`)

Incompatible with `-s`.

`-w name_of_generated_type-safe_API_wrapper_class_and_assembly`

Optional. If specified, overrides default name of generated type-safe API class and assembly

Incompatible with `-c`.

Outputs

`ComponentInterface.dll`

.NET binary containing type-safe API class. Requires `ComponentNative.dll`, `Interface.dll` and `MWArray.dll`

`ComponentInterface.cs`

Optional output, produced by `-s` and `-k`

Examples

```
ntswrap.exe -c AddOneComp.Mechanism
```

```
-i IAddOne  
-a IAddOne.dll
```

Issuing this command generates a type-safe API for the MATLAB Builder NE class `Mechanism` in the namespace `AddOneCompNative`. By default, `ntswrap` compiles the source code into an assembly `MechanismIAddOne.dll`.

Create and Install COM Components

- “Build a Deployable COM Component” on page 13-2
- “Package a Deployable COM Component” on page 13-3
- “Embedded Deployable Archives” on page 13-5
- “Use the Command-Line Interface” on page 13-6
- “Install COM Components on a Target Computer” on page 13-9

Build a Deployable COM Component

See “Create a .NET Assembly From MATLAB Code” and select the Generic COM Component as the compilation target.

Package a Deployable COM Component

Add-in and COM Component Registration

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

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

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

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

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

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

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

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

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

1 Unregister the component with this command:

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

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

```
mwregsvr my_dll.dll
```

Embedded Deployable Archives

As of R2008b, the MATLAB Builder NE product now embeds the deployable 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 deployable archive (the default behavior before R2008b), see “MATLAB Runtime Component Cache and Deployable Archive Embedding”.

Use the Command-Line Interface

A MATLAB class cannot be directly compiled into a COM object. You can, however, use a user-generated class inside a MATLAB file and build a COM object from that file. You can use the MATLAB command-line interface instead of the library compiler app 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

Action to Perform	mcc Option to Use	Description
Create component that has one class.	<code>-W com</code>	The <code>W</code> option with <code>com</code> as the type controls the generation of wrapper files, which you can use to support components.
	Syntax	
	<pre>mcc -W 'com:<component_name>[,<class_name>[,<major>.<minor>]]'</pre> <p>An unspecified <code><class_name></code> defaults to <code><component_name></code>, and an unspecified version number defaults to the latest version built or 1.0, if there is no previous version.</p>	
	Example	
<pre>mcc -W 'com:mycomponent,myclass,1.0' -T link:lib foo.m bar.m</pre> <p>The example creates a COM component called <code>mycomponent</code>, which contains a single COM class named <code>myclass</code> with methods <code>foo</code> and <code>bar</code>, and a version of 1.0.</p>		

Action to Perform	mcc Option to Use	Description
Add additional classes to a COM component.	Not needed	<p>A separate COM named <code><class_name></code> is created for each class argument that is passed.</p> <p>Following the <code><class_name></code> parameter is a comma-separated list of source files that are encapsulated as methods for the class.</p>
	Syntax	<pre>class{<class_name>:[file, [file,...]]}</pre>
	Example	<pre>mcc -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m class{myclass2:foo2.m, bar2.m}</pre> <p>The example creates a COM component named <code>mycomponent</code> with two classes: <code>myclass</code> has methods <code>foo</code> and <code>bar</code>, and <code>myclass2</code> has methods <code>foo2</code> and <code>bar2</code>. The version is version 1.0.</p>
Simplify the command-line input for components.	-B ccom:	Uses the bundle file.
	Syntax	<pre>mcc -B '<filename>'[:<a1>,<a2>,...,<an>]</pre>
	Example	<pre>mcc -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m</pre>
Control how each COM class uses the MATLAB runtime.	-S	<p>By default, a new MATLAB runtime instance is created for each instance of each COM class in the component. Use <code>-S</code> to change the default.</p> <p>This option tells the builder to create a single MATLAB runtime at the time when the first COM class is instantiated. This runtime is reused and shared among all subsequent class instances, resulting in more efficient memory usage and eliminating the MATLAB runtime startup cost in each subsequent class instantiation.</p> <p>When using <code>-S</code>, note that all class instances share a single MATLAB workspace and share global</p>

Action to Perform	mcc Option to Use	Description
		<p>variables in the MATLAB files used to build the component. Therefore, properties of a COM class behave as static properties instead of instance-wise properties.</p> <hr/> <p>Note: The default behavior dictates that a new MATLAB runtime be created for each instance of a class, so when the class is destroyed, the runtime 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.</p> <hr/> <p>Example</p> <pre>mcc -S -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m</pre> <p>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.</p> <p>When multiple instances of this class are instantiated in an application, only one MATLAB runtime is initialized, and it is shared by each instance.</p>
<p>Create subfolders needed for deployment and copy associated files to them.</p>	<p><code>-d</code></p> <hr/> <p>Syntax</p> <p><code>-d <i>foldername</i></code></p>	<p>The <code>\src</code> and <code>\distrib</code> subfolders are needed to package components.</p>

Install 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 runtime as described in “Distributing MATLAB Code Using the MATLAB Runtime”.
- 2** Build and package as described in “Build a Deployable COM Component” on page 13-2 and “Package a Deployable COM Component” on page 13-3.
- 3** Copy the package to the target computer and run the package.
- 4** From a Windows command prompt on the target system, navigate to the folder where you saved the package. If you use the command `dir`, you should see the `.dll` created for your COM object. You will need to register the `.dll` manually using the command `regsvr32`, as follows:

```
regsvr32 myCom_1_0.dll
```


Programming with COM Components Created by the MATLAB Builder NE Product

- “General Techniques” on page 14-2
- “Registering and Referencing the Utility Library” on page 14-3
- “Creating an Instance of a Class in Microsoft Visual Basic” on page 14-4
- “Calling the Methods of a Class Instance” on page 14-7
- “Calling a COM Object in a Visual C++ Program” on page 14-10
- “Using a COM Component in a .NET Application” on page 14-11
- “Adding Events to COM Objects” on page 14-12
- “Passing Arguments ” on page 14-16
- “Using Flags to Control Array Formatting and Data Conversion” on page 14-19
- “Using MATLAB Global Variables in Microsoft Visual Basic” on page 14-26
- “Blocking Execution of a Console Application That Creates Figures” on page 14-29
- “MATLAB Runtime Run-Time Options” on page 14-32
- “Sharing a MATLAB Runtime Instance in COM or Java Applications” on page 14-33
- “Obtaining Registry Information” on page 14-35
- “Handling Errors During a Method Call” on page 14-37

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 14-3
 - “Creating an Instance of a Class in Microsoft Visual Basic” on page 14-4
- Call methods of the component class
 - “Calling the Methods of a Class Instance” on page 14-7
 - “Calling a COM Object in a Visual C++ Program” on page 14-10
 - “Adding Events to COM Objects” on page 14-12
 - “Obtaining Registry Information” on page 14-35
- Deal with data conversion and parameter passing
 - “Passing Arguments ” on page 14-16
 - “Using Flags to Control Array Formatting and Data Conversion” on page 14-19
 - “Using MATLAB Global Variables in Microsoft Visual Basic” on page 14-26
- Process errors
 - “Handling Errors During a Method Call” on page 14-37

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. `mwregsvr.exe` is supplied with the MATLAB runtime.

Creating an Instance of a Class in Microsoft Visual Basic

In this section...

“Advantages and Disadvantages” on page 14-4

“CreateObject Function” on page 14-4

“Microsoft Visual Basic New Operator” on page 14-5

“Advantages of Each Technique” on page 14-5

“Declaring a Reusable Class Instance” on page 14-6

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 14-10.

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 14-7

“Variant” on page 14-8

“Examples of Passing Input and Output Parameters” on page 14-8

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 16-20 for more details and examples of the standard mapping from MATLAB functions to COM class method calls.

Variant

All input and output arguments are typed as `Variant`, the default Visual Basic data type. The `Variant` type can hold any of the basic Visual Basic types, arrays of any type, and object references. See “Data Conversion” on page 16-8 for details about the conversion of any basic type to and from MATLAB data types.

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

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

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

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

Use the COM object you have created as follows:

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

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

// include 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
    mycomponentclass *pMycomponentclass;
    hr=CoCreateInstance
        (CLSID_mycomponentclass, NULL, CLSCTX_INPROC_SERVER,
         IID_mycomponentclass, (void **)&pMycomponentclass);
    // 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=(pMycomponentclass -> 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 14-11
“Program Listings” on page 14-11

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.

Program Listings

In *matlabroot*\toolbox\dotnetbuilder\Examples\VSVersion\COM
\CurveFitExample\:

C# Example

CurveFitCSharpApp\CurveFitApp.cs

Visual Basic Example

CurveFitVBApp\CurveFitApp.vb

Adding Events to COM Objects

In this section...

“MATLAB Language Pragma” on page 14-12

“Using a Callback with a Microsoft Visual Basic Event” on page 14-13

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;
```

progress.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, CDb1(N), CDb1(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 14-16

“Creating and Using a varargin Array in Microsoft Visual Basic Programs” on page 14-16

“Creating and Using varargout in Microsoft Visual Basic Programs” on page 14-17

“Passing an Empty varargin From Microsoft Visual Basic Code” on page 14-17

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 16-20 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.

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 14-19

“Array Formatting Flags” on page 14-19

“Using Array Formatting Flags” on page 14-20

“Using Data Conversion Flags” on page 14-22

“Special Flags for Some Microsoft Visual Basic Types” on page 14-24

Overview

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

In some cases it is not possible to match the two kinds of arguments exactly; for example, when existing MATLAB code is used in conjunction with a third-party product such as Microsoft Excel. For these and other cases, the 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
<code>InputArrayFormat</code>	<code>mwArrayFormatMatrix</code> (default)	MATLAB matrix from general Variant data.

Name of Flag	Possible Values of Flag	Results of Conversion
	mwArrayFormatCell	MATLAB cell array from general Variant data.
	Array data from an Excel range is coded in Visual Basic as an array of Variant. Since MATLAB functions typically have matrix arguments, using the default setting makes sense when you are dealing with data from Excel.	
OutputArrayFormat	mwArrayFormatAsIs	Array of Variant
	Converts arrays according to the default conversion rules listed in MATLAB to COM VARIANT Conversion Rules.	
	mwArrayFormatMatrix	A Variant containing an array of a basic type.
	mwArrayFormatCell	MATLAB cell array from general Variant data.
AutoSizeOutput	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- <i>n</i> 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 21 22	11 12 21 22
Internal structure in Visual Basic	2-by-2 array of <code>Variant</code> . Each <code>variant</code> is a 1-by-1 array of <code>Double</code> .	1-by-1 <code>Variant</code> , which contains a 2-by-2 array of <code>Double</code>
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 <code>double</code> .	2-by-2 matrix. Each element is a <code>Double</code> .

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 14-29

“Using MCRWaitForFigures to Block Execution” on page 14-29

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 library compiler app, you can create the component by issuing the following command at the MATLAB prompt:

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

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

```
// Include 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.

MATLAB Runtime Run-Time Options

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

Pass these options with `mcc`

What MATLAB Runtime Options are Supported for COM?

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

How Do I Specify MATLAB Runtime Options?

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

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

Sharing a MATLAB Runtime Instance in COM or Java Applications

In this section...

“What Is a Singleton MATLAB Runtime?” on page 14-33

“Advantages and Disadvantages of Using a Singleton” on page 14-33

“Which Products Support Singleton MATLAB Runtime and How Do I Create a Singleton?” on page 14-34

What Is a Singleton MATLAB Runtime?

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

Advantages and Disadvantages of Using a Singleton

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

When You Should Use a Singleton

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

- Utilize system memory more efficiently
- Decrease MATLAB runtime start-up or initialization time
- Promote reuse of your application code base

When You Might Avoid Using a Singleton

Situations where using a singleton may not benefit you include:

- Running applications with a large number of global variables. This can promote crosstalk which can eventually impact performance.
- Your installation runs many different versions of MATLAB, for testing purposes.
- Your installation has a relative

- On the library compiler , select **Exclusive MCR** under **Additional Runtime Settings**.

y small number of users and is not overly concerned with performance.

Which Products Support Singleton MATLAB Runtime and How Do I Create a Singleton?

Singleton MATLAB runtime is only supported by the following products on these specific targets:

Product	Target supported by Singleton MATLAB Runtime	Create a Singleton MATLAB Runtime by....
MATLAB Builder EX	COM component	Default behavior for target is singleton runtime. You do not need to perform other steps.
MATLAB Builder NE	.NET assembly	Default behavior for target is singleton runtime. You do not need to perform other steps.
MATLAB Builder NE	COM component	<ul style="list-style-type: none"> • Using the shared library compiler app, click Settings and add -S to the Additional flags to pass to mcc field. • Using mcc pass the -S flag.
MATLAB Builder JA	Java packages	

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 error handling.

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 15-7
- “Univariate Interpolation Example” on page 15-19
- “Matrix Calculator Example” on page 15-25
- “Curve Fitting Example” on page 15-33
- “Bouncing Ball Simulation Example” on page 15-39

Magic Square Example

In this section...

“Example Overview” on page 15-2

“Creating the MATLAB File” on page 15-2

“Using the Library Compiler App to Create and Build the Project” on page 15-2

“Creating the Microsoft Visual Basic Project” on page 15-3

“Creating the User Interface” on page 15-3

“Creating the Executable in Microsoft Visual Basic” on page 15-5

“Testing the Application” on page 15-5

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 Library Compiler App to Create and Build the Project

- 1 While in MATLAB, open the Library Compiler app.
- 2 Select **Generic COM Component** as the application type.

- 3 Add `magicsquare.m` to the list of exported functions.
`magicsquare.m` is located in the `MagicDemoComp` folder.
- 4 Click the **Package** button.

Creating the Microsoft Visual Basic Project

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

- 1 Start Visual Basic.
- 2 In the New Project dialog box, select **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	Displays the magic square.

Control Type	Control Name	Properties	Purpose
		LabelEdit = lvwManual View = lvwReport	

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.

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

Creating an Excel Add-in: Spectral Analysis Example

In this section...

“Example Overview” on page 15-7

“Building the Component” on page 15-7

“Integrating the Component with VBA” on page 15-9

“Creating the Microsoft Visual Basic Form” on page 15-10

“Adding the Spectral Analysis Menu Item to Microsoft Excel” on page 15-15

“Saving the Add-in” on page 15-16

“Testing the Add-in” on page 15-16

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 than zero');  
        return;  
    end  
    fftdata = fft(data);  
    freq = (0:length(fftdata)-1)/(length(fftdata)*interval);  
    powerspect = abs(fftdata)/(sqrt(length(fftdata)));
```

`plotfft.m:`

```
function [fftdata, freq, powerspect] = plotfft(data, interval)  
    [fftdata, freq, powerspect] = computefft(data, interval);  
    len = length(fftdata);  
    if (len <= 0)  
        return;  
    end  
    t = 0:interval:(len-1)*interval;  
    subplot(2,1,1), plot(t, data)  
    xlabel('Time'), grid on  
    title('Time domain signal')  
    subplot(2,1,2), plot(freq(1:len/2), powerspect(1:len/2))  
    xlabel('Frequency (Hz)'), grid on  
    title('Power spectral density')
```

To 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.
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.

Control Type	Control Name	Properties	Purpose
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.
RefEdit	refedtPowSpect		Selects the output range for power spectral density of input data.
CommandButton	btnOK	Caption = OK Default = True	Executes the function and closes the dialog box.
CommandButton	btnCancel	Caption = Cancel 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 (theFFTDData.Real) Then
    If IsObject(theFFTDData.Real) And TypeOf theFFTDData.Real Is Range Then
        refedtReal.Text = theFFTDData.Real.Address
    End If
    End If
    If Not IsEmpty (theFFTDData.Imag) Then
    If IsObject(theFFTDData.Imag) And TypeOf theFFTDData.Imag Is Range Then
        refedtImag.Text = theFFTDData.Imag.Address
    End If
    End If
    If Not PowerSpect Is Nothing Then
        refedtPowSpect.Text = PowerSpect.Address
    End If
    chkPlot.Value = bPlot
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

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

    If theFourier Is Nothing Or theFFTDData Is Nothing Then GoTo Exit_Form
    On Error Resume Next
    'Process inputs
    Set R = Range(refedtInput.Text)
    If Err <> 0 Then
        MsgBox ("Invalid range entered for Input Data")
        Exit Sub
    End If
    Set InputData = R
    Interval = CDb1(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
        theFFTDData.Real = R
    End If
    Set R = Range(refedtImag.Text)
    If Err = 0 Then
        theFFTDData.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 $\text{SIN}(2*\text{PI}() * 15 * \text{A1}) + \text{SIN}(2*\text{PI}() * 40 * \text{A1}) + \text{RAND}()$.
- 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.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	0	0.201422	0	500.5243	0	15.82797									
2	0.01	2.364546	0.1	8.44324	1.112117	0.269305									
3	0.02	0.424141	0.2	-0.28779	2.589662	0.062396									
4	0.03	1.956734	0.3	8.773018	2.110449	0.285342									
5	0.04	-1.06567	0.4	-10.035	-2.49211	0.326975									
6	0.05	-0.50068	0.5	-1.59236	-2.69405	0.088962									
7	0.06	0.992995	0.6	4.42359	9.082895	0.319479									
8	0.07	-0.2288	0.7	-13.0441	-0.19807	0.412538									
9	0.08	2.495782	0.8	-7.36004	2.259597	0.243467									
10	0.09	0.853563	0.9	-6.57995	-12.3651	0.442935									
11	0.1	0.33184	1	3.447282	4.201153	0.171853									
12	0.11	-0.00625	1.1	-10.3228	-5.27412	0.366574									
13	0.12	-1.36424	1.2	9.284435	4.187427	0.32208									
14	0.13	1.259695	1.3	-11.5624	6.969041	0.426915									
15	0.14	0.204847	1.4	5.8311	-1.10339	0.187668									
16	0.15	1.010208	1.5	3.255038	-3.99248	0.162896									
17	0.16	1.640045	1.6	1.436592	11.66599	0.371698									
18	0.17	-0.44132	1.7	-3.00437	-3.39444	0.143347									
19	0.18	0.211505	1.8	2.89991	5.71643	0.202699									
20	0.19	-0.99357	1.9	1.121279	8.400998	0.268019									
21	0.2	0.379924	2	-2.80552	-6.592	0.226551									
22	0.21	1.83771	2.1	-8.45165	-1.5761	0.271872									
23	0.22	0.368155	2.2	-3.37307	-10.252	0.341293									
24	0.23	2.117552	2.3	-5.24299	-6.97011	0.27581									
25	0.24	-0.66651	2.4	0.925776	-5.9012	0.188895									
26	0.25	-0.33152	2.5	3.100077	4.991781	0.185818									
27	0.26	0.394927	2.6	-3.16232	0.028866	0.100006									
28	0.27	0.143179	2.7	9.481148	-3.10764	0.315515									
29	0.28	2.740805	2.8	5.448927	-1.52843	0.178961									
30	0.29	0.230976	2.9	1.95382	-11.3264	0.363461									
31	0.3	0.913667	3	-1.32565	-3.82213	0.12793									
32	0.31	0.117747	3.1	1.148378	-9.15371	0.291735									
33	0.32	-1.57611	3.2	11.54045	-5.22124	0.400554									
34	0.33	1.577908	3.3	-3.13453	1.159207	0.105684									
35	0.34	0.373403	3.4	1.005112	0.107201	0.031965									
36	0.35	1.405944	3.5	0.69275	0.803781	0.033555									
37	0.36	2.163784	3.6	-0.29716	-13.7448	0.434751									
38	0.37	-0.79952	3.7	-4.04466	-5.47217	0.215183									
39	0.38	0.31803	3.8	-5.84649	0.003747	0.184882									
40	0.39	-0.80853	3.9	-2.43879	6.877807	0.230764									
41	0.4	0.597445	4	1.717283	1.174817	0.065797									
42	0.41	2.119417	4.1	1.218546	-14.3198	0.454469									
43	0.42	0.01473	4.2	-0.52696	17.72755	0.560842									
44	0.43	2.131939	4.3	-2.70571	0.383641	0.086418									
45	0.44	-0.37325	4.4	1.248446	10.43094	0.332209									
46	0.45	-0.17902	4.5	-4.42196	-16.158	0.529749									

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

Univariate Interpolation Example

In this section...

“Example Overview” on page 15-19

“Using the Library Compiler App to Create and Build the Component” on page 15-19

“Using the Component in Microsoft Visual Basic” on page 15-19

“Creating the Microsoft Visual Basic Form” on page 15-20

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 Library Compiler App to Create and Build the Component

- 1 At the MATLAB command prompt, change folders to your work folder.
- 2 Open the Library Compiler app.
- 3 Select **Generic COM Component** as the application type.
- 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 **Package** button.

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
 MComUtil 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 GridLines = True LabelEdit = lvwAutomatic 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 GridLines = True LabelEdit = lvwAutomatic 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 GridLines = True LabelEdit = lvwAutomatic View = lvwReport	Interpolation points
CommandButton	cmdEvaluate	Caption = Evaluate Default = True	Executes the function
CommandButton	cmdCancel	Caption = Cancel 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.MWFFlags.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 15-25

“Building 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 15-26

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 eigen values 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 Open the Library Compiler app.
- 3 Select **Generic COM Component** as the application type.
- 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 **Package** button.

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.
- 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

Control Type	Control Name	Properties	Purpose
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
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 Default = True	Executes function
CommandButton	cmdCancel	Caption = Cancel 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
    Next RowCount

```

```

        ' 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

```

```
        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 15-33

“Building the Component” on page 15-33

“Using the Component in Microsoft Visual Basic” on page 15-33

“Creating the Microsoft Visual Basic Form” on page 15-34

Example Overview

This example demonstrates the optimal fitting of a nonlinear function to a set of data, using the curve-fitting example `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 Open the Library Compiler app.
- 3 Select **Generic COM Component** as the application type.
- 4 Add `fitfun.m` and `fitdemo.m` from the folder *matlabroot/CurveFitDemo*.
- 5 Click the **Package** button in the toolbar.

Using the Component in Microsoft Visual Basic

You can call the component from any application that supports COM.

Open CurveFitComp.vcproj or 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 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 **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* Caption = Input Data Points	Groups all input controls.
Frame	frmOutput	Name = frmOutput*	Groups all output controls.

Control Type	Control Name	Properties	Purpose
		Caption = Output Coefficients	
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.
ListView	lstXData	Name = lstXData GridLines = TrueLabel Edit = lvwAutomatic View = lvwReport	X-data values. Set the view type to lvwReport to enable user to add data to the list view.
ListView	lstYData	Name = lstYData GridLines = TrueLabel Edit = lvwAutomatic 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.

Control Type	Control Name	Properties	Purpose
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.
TextBox	txtLambda2	Text =	Displays value of lambda 2 as calculated by the COM module.
CommandButton	cmdEvaluate	Caption = Evaluate Default = True	Executes function.
CommandButton	cmdCancel	Caption = Cancel 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 executed 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 run-time 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)
    End For
End Sub
```

```

    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 15-39

“Building the Component” on page 15-39

“Using the Component in Microsoft Visual Basic” on page 15-40

“Creating the Microsoft Visual Basic Form” on page 15-40

Example Overview

This example is adapted from the `ballode` example provided with the MATLAB product. It demonstrates repeated event location, where the conditions are changed after each terminal event.

This example 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 the example completes.

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 Open the Library Compiler app.
- 3 Select **Generic COM Component** as the application type.
- 4 Locate your work folder, navigate to *matlabroot/BallODE*, and add `ballode.m` to the project.
- 5 Build the component by clicking the **Package** button.

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	frmBall1ode	Caption = Bouncing Ball ODE	Container for all components.

Control Type	Control Name	Properties	Purpose
Frame	frmInput	Name = frmInput* Caption = Input Data Points	Groups all input controls.
Frame	frmOutput	Name = frmOutput* 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.
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 example is completed.
ListView	lstBounce	Name = lstBounce GridLines = True LabelEdit = lvwManual View = lvwReport	Displays the time stamp when ball bounces off the ground.

Control Type	Control Name	Properties	Purpose
CommandButton	cmdEvaluate	Caption = Evaluate Default = True	Executes the function.
CommandButton	cmdCancel	Caption = Cancel 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
```


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 16-8
- “Calling Conventions” on page 16-20

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 16-2

“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 16-6

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 MATLAB runtime, registers self-registering DLLs. For example, to register a component called `mycomponent_1_0.dll`, issue this command at the DOS command prompt:

```
mwregsvr mycomponent_1_0.dll
```

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

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

unregisters the component.

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

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

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

Globally Unique Identifier

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

The 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
<code>HKEY_CLASSES_ROOT\CLSID</code>	Information about COM classes on the system. Each component creates a new key under <code>HKEY_CLASSES_ROOT\CLSID</code> 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.
<code>HKEY_CLASSES_ROOT\Interface</code>	Information about COM interfaces on the system. Each component creates a new key under <code>HKEY_CLASSES_ROOT\Interface</code> for each interface it defines.

Key	Definition
	This key has the value of the GUID assigned to the interface and contains subkeys with information about the interface.
HKEY_CLASSES_ROOT\TypeLib	Information about type libraries on the system. Each component creates a key for its type library with the value of the GUID assigned to it. Under this key a new key is created for each version of the type library. Therefore, new versions of type libraries with the same name reuse the original GUID but create a new subkey for the new version.
HKEY_CLASSES_ROOT\<>ProgID>, HKEY_CLASSES_ROOT\<>VerIndProgID>	These two keys are created for the component's Program ID and Version Independent Program ID. These keys are constructed from strings of the following forms: <i>component-name.class-name</i> <i>component-name.class-name</i> <i>version-number</i> 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 16-8

“Array Formatting Flags” on page 16-17

“Data Conversion Flags” on page 16-18

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

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

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

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

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

VARIANT Type Codes Supported

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

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

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

MATLAB to COM VARIANT Conversion Rules

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
cell	A 1-by-1 cell array converts to a single VARIANT with a type conforming to the conversion rule for the MATLAB data type of the cell contents.	A multidimensional cell array converts to a VARIANT of type VT_VARIANT VT_ARRAY with the type of each array member conforming to the conversion rule for the MATLAB data type of the corresponding cell.	
structure	VT_DISPATCH	VT_DISPATCH	A MATLAB struct array is converted to an MWStruct object. (See “Class MWStruct” on page 17-18.) This object is passed as a VT_DISPATCH type.
char	A 1-by-1 char matrix converts to a VARIANT of type VT_BSTR with string length = 1.	A 1-by-L char matrix is assumed to represent a string of length L in MATLAB. This case converts to a VARIANT of type VT_BSTR with a string length = L. char matrices of more than one row, or of a higher dimensionality convert to a VARIANT	Arrays of strings are not supported as char matrices. To pass an array of strings, use a cell array of 1-by-L char matrices.

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
		of type VT_BSTR VT_ARRAY. Each string in the converted array is of length 1 and corresponds to each character in the original matrix.	
sparse	VT_DISPATCH	VT_DISPATCH	A MATLAB sparse array is converted to an MWSparse object. (See “Class MWSparse” on page 17-30.) 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 17-27
single	A real 1-by-1 single matrix converts to a VARIANT of type VT_R4. A complex 1-by-1 single matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional single matrix converts to a VARIANT of type VT_R4 VT_ARRAY. A complex multidimensional single matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class.
int8	A real 1-by-1 int8 matrix converts to a VARIANT of type VT_I1. A complex 1-by-1	A real multidimensional int8 matrix converts to a VARIANT of type VT_I1 VT_ARRAY.	Complex arrays are passed to and from compiled MATLAB

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

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
	1-by-1 int32 matrix converts to a VARIANT of type VT_DISPATCH.	of type VT_I4 VT_ARRAY. A complex multidimensional int32 matrix converts to a VARIANT of type VT_DISPATCH.	functions using the MWComplex class.
uint32	A 1-by-1 uint32 matrix converts to a VARIANT of type VT_UI4. A complex 1-by-1 uint32 matrix converts to a VARIANT of type VT_DISPATCH.	A multidimensional uint32 matrix converts to a VARIANT of type VT_UI4 VT_ARRAY. A complex multidimensional uint32 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled MATLAB functions using the MWComplex class.
Function handle	VT_EMPTY	VT_EMPTY	Not supported
Java class	VT_EMPTY	VT_EMPTY	Not supported
User class	VT_EMPTY	VT_EMPTY	Not supported
logical	VT_Boo1	VT_Boo1 VT_ARRAY	

COM VARIANT to MATLAB Conversion Rules

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

VARIANT Type	MATLAB Data Type (Scalar or Array Data)	Comments
VT_R8	double	
VT_CY	double	
VT_BSTR	char	A VARIANT of type VT_BSTR converts to a 1-by-L MATLAB char array, where L = the length of the string to be converted. A VARIANT of type VT_BSTR VT_ARRAY converts to a MATLAB cell array of 1-by-L char arrays.
VT_ERROR	int32	
VT_DATE	double	VARIANT dates are stored as doubles starting at midnight Dec. 31, 1899. MATLAB dates are stored as doubles starting at 0/0/00 00:00:00. Therefore, a VARIANT date of 0.0 maps to a MATLAB numeric date of 693960.0. VARIANT dates are converted to MATLAB double types and incremented by 693960.0. VARIANT dates can be optionally converted to strings. See “Data Conversion Flags” on page 16-18 for more information on type coercion.
VT_INT	int32	
VT_UINT	uint32	
VT_DECIMAL	double	
VT_BOOL	logical	
VT_DISPATCH	Varies	IDispatch* pointers are treated within the context of what they point to. Objects must be supported types with known

VARIANT Type	MATLAB Data Type (Scalar or Array Data)	Comments
		<p>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.</p> <p>Currently, support exists for Excel Range objects as well as the builder types MWStruct, MWComplex, MWSparse, and MWArg. See “Referencing Utility Classes” on page 17-2 for information on the builder types to use with COM components.</p>
<i>anything</i> VT_BYREF	<i>Varies</i>	<p>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.</p>
<i>anything</i> VT_ARRAY	<i>Varies</i>	<p>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.</p>

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 and COM VARIANT to MATLAB Conversion Rules. In some cases this is not possible, for example, when existing MATLAB code is used in conjunction with a third-party product like Excel.

The following table shows the array formatting flags.

Array Formatting Flags

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

Flag	Description
	array. When this flag is greater than zero, e.g., equal to N, the formatting rule attempts to apply itself to the Nth level of nesting.
OutputArrayFormat	Defines the array formatting rule used on output arrays. An output array is a MATLAB array, created by the compiled COM object, sent as an output parameter from a method call to the client. The values for this flag, <code>mwArrayFormatAsIs</code> , <code>mwArrayFormatMatrix</code> , and <code>mwArrayFormatCell</code> , cause the same behavior as the corresponding <code>InputArrayFormat</code> flag values.
OutputArrayIndFlag	(Applies to nested cell arrays only.) Output array indirection level used with the <code>OutputArrayFormat</code> flag. This flag works exactly like <code>InputArrayIndFlag</code> .
AutoSizeOutput	(Applies to Excel ranges only.) When the target output from a method call is a range of cells in an Excel worksheet and the output array size and shape is not known at the time of the call, set this flag to <code>True</code> to resize each Excel range to fit the output array.
TransposeOutput	Set this flag to <code>True</code> to transpose the output arguments. Useful when calling a MATLAB Builder 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 16-18
- “InputDateFormat” on page 16-19
- “OutputAsDate As Boolean” on page 16-19
- “DateBias As Long” on page 16-19

CoerceNumericToType

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

VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R8, VT_CY, VT_DECIMAL, VT_INT, VT_UINT, VT_ERROR, VT_BOOL, and VT_DATE. Valid values for this flag are `mwTypeDefault`, `mwTypeChar`, `mwTypeDouble`, `mwTypeSingle`, `mwTypeLogical1`, `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 16-8.

InputDateFormat

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

OutputAsDate As Boolean

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

DateBias As Long

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

Calling Conventions

In this section...

- “Producing a COM Class” on page 16-20
- “IDL Mapping” on page 16-21
- “Microsoft Visual Basic Mapping” on page 16-21

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.

Code	Sample
Generic MATLAB Code	<code>function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)</code>
IDL Code	<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>

Code	Sample
Visual Basic Code	<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>

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 16-8 lists the rules for conversion between MATLAB arrays and `COM VARIANT`s.

Microsoft Visual Basic Mapping

Microsoft Visual Basic provides native support for `COM Variants` with the `Variant` type, as well as implicit conversions for all Visual Basic primitive types to and from

Variants. In general, arrays/scalars of any Visual Basic primitive 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
- “Class MWUtil” on page 17-3
- “Class MWFlags” on page 17-12
- “Class MWStruct” on page 17-18
- “Class MWField” on page 17-25
- “Class MWComplex” on page 17-27
- “Class MWSparse” on page 17-30
- “Class MWArg” on page 17-34
- “Enum mwArrayFormat” on page 17-36
- “Enum mwDataType” on page 17-37
- “Enum mwDateFormat” on page 17-38

Referencing Utility Classes

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

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

```
mwregsvr mwcomutil.dll
```

The `MWComUtil` library includes seven classes and three enumerated types. Before using these types, you must make explicit references to the `MWComUtil` type libraries in the Microsoft Visual Basic IDE. To do this select **Tools > References** from the main menu of the Visual Basic Editor. The References dialog box appears with a scrollable list of available type libraries. From this list, select **MWComUtil 1.0 Type Library** and click **OK**.

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

Class MWUtil

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

In this section...
“Sub MWInitApplication(pApp As Object)” on page 17-3
“Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])” on page 17-5
“Function IsMCRJVMEEnabled() As Boolean” on page 17-6
“Function IsMCRInitialized() As Boolean” on page 17-6
“Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])” on page 17-7
“Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoSize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])” on page 17-8
“Sub MWDate2VariantDate(pVar)” on page 17-10

The function prototypes use Visual Basic syntax.

Sub MWInitApplication(pApp As Object)

Initializes the library with the current instance of Microsoft Excel.

Parameters

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

Return Value

None.

Remarks

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

Example

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

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

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

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

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

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

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

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

Sub MWInitApplicationWithMCROptions(pApp As Object, [mcrOptionList])

Start MATLAB runtime with MATLAB runtime options. Similar to “mclInitializeApplication”.

Parameters

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

Return Value

None.

Remarks

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

Example

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

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

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

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

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

Function IsMCRJVMEabled() As Boolean

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

Parameters

None.

Return Value

Boolean

Function IsMCRInitialized() As Boolean

Returns true if MATLAB runtime is initialized; otherwise returns true

Parameters

None.

Return Value

Boolean

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

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

Parameters

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

Return Value

None.

Remarks

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

Example

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

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

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

Function mysum(Optional V0 As Variant, _

```

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

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

```

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

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

Parameters

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

Argument	Type	Description
bAutoSize	Boolean	Optional auto-resize flag. If this flag is 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 *i*th vector equal to *i*. Assume that this function is included in a class named **myclass** that is included in a component named **mycomponent** with a version of 1.0. The Visual Basic subroutine takes no arguments and places the results into Excel columns starting

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

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

Sub MWDate2VariantDate(pVar)

Converts output dates from MATLAB to Variant dates.

Parameters

Argument	Type	Description
pVar	Variant	Variant to be converted

Return Value

None.

Remarks

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

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

Example

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

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

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

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

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

Class MWFlags

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

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

Property ArrayFormatFlags As MWArrayFormatFlags

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

- “Property InputArrayFormat As mwArrayFormat” on page 17-12
- “Property InputArrayIndFlag As Long” on page 17-13
- “Property OutputArrayFormat As mwArrayFormat” on page 17-13
- “Property OutputArrayIndFlag As Long” on page 17-14
- “Property AutoResizeOutput As Boolean” on page 17-14
- “Property TransposeOutput As Boolean” on page 17-14

Property InputArrayFormat As mwArrayFormat

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

Array Formatting Rules for Input Arrays

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

Property `InputArrayIndFlag` As Long

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

Property `OutputArrayFormat` As `mwArrayFormat`

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

Array Formatting Rules for Output Arrays

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

Property `OutputArrayIndFlag` As Long

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

Property `AutoSizeOutput` As Boolean

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

Property `TransposeOutput` As Boolean

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

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

Property `DataConversionFlags` As `MWDataConversionFlags`

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

- “Property `CoerceNumericToType` As `mwDataType`” on page 17-15
- “Property `DateBias` As `Long`” on page 17-15
- “Property `InputDateFormat` As `mwDateFormat`” on page 17-16
- “Property `OutputAsDate` As `Boolean`” on page 17-17
- “Property `ReplaceMissing` As `mwReplaceMissingData`” on page 17-17

Property `CoerceNumericToType` As `mwDataType`

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

Property `DateBias` As `Long`

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

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

```
function p = myprimes(n)
if length(n)~=1, error('N must be a scalar'); end
if n < 2, p = zeros(1,0); return, end
p = 1:2:n;
q = length(p);
p(1) = 2;
for k = 3:2:sqrt(n)
```

```

        if p((k+1)/2)
            p(((k*k+1)/2):k:q) = 0;
        end
    end
    p = (p(p>0));

```

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

```

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

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

```

Property `InputDateFormat` As `mwDateFormat`

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

Conversion Rules for Input Dates

Value	Behavior
<code>mwDateFormatNumeric</code>	Convert dates to numeric values as indicated by the rule listed in “Data Conversion Rules”.

Value	Behavior
mwDateFormatString	Convert input dates to strings.

PropertyOutputAsDate As Boolean

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

ReplaceMissing As mwReplaceMissingData

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

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

By default, the value is **mwReplaceZero**.

Sub Clone(ppFlags As MWFlags)

Creates a copy of an **MWFlags** object.

Parameters

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

Return Value

None

Remarks

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

Class MWStruct

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

In this section...

“Sub `Initialize`([`varDims`], [`varFieldNames`])” on page 17-18

“Property `Item`([`i0`], [`i1`], ..., [`i31`]) As `MWField`” on page 17-19

“Property `NumberOfFields` As Long” on page 17-22

“Property `NumberOfDims` As Long” on page 17-22

“Property `Dims` As Variant” on page 17-22

“Property `FieldNames` As Variant” on page 17-22

“Sub `Clone`(`ppStruct` As `MWStruct`)” on page 17-23

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

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

Parameters

Argument	Type	Description
<code>varDims</code>	Variant	Optional array of dimensions
<code>varFieldNames</code>	Variant	Optional array of field names

Return Value

None.

Remarks

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

Example

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

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

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

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

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

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

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

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

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

Parameters

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

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

Remarks

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

- Field name only

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

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

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

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

- Single index and field name

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

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

- All indices and field name

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

```

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

```

- Array of indices and field name

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

```

Dim Index(1 To 2) As Integer

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

```

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

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

```

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

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

```

The last statement resolves to

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

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

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

- Field names are case sensitive.

Property NumberOfFields As Long

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

Property NumberOfDims As Long

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

Property Dims As Variant

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

Property FieldNames As Variant

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

Example

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

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

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

```

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

```

Sub Clone(ppStruct As MWStruct)

Creates a copy of an MWStruct object.

Parameters

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

Return Value

None

Remarks

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

Example

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

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

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

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

Class MWField

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

In this section...
“Property Name As String” on page 17-25
“Property Value As Variant” on page 17-25
“Property MWFlags As MWFlags” on page 17-25
“Sub Clone(ppField As MWField)” on page 17-25

Property Name As String

The name of the field (read only).

Property Value As Variant

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

Property MWFlags As MWFlags

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

Sub Clone(ppField As MWField)

Creates a copy of an `MWField` object.

Parameters

Argument	Type	Description
<code>ppField</code>	<code>MWField</code>	Reference to an uninitialized <code>MWField</code> object to receive the copy

Return Value

None.

Remarks

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

Class MWComplex

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

In this section...

“Property `Real As Variant`” on page 17-27

“Property `Imag As Variant`” on page 17-27

“Property `MWFlags As MWFlags`” on page 17-28

“Sub `Clone(ppComplex As MWComplex)`” on page 17-28

Property `Real As Variant`

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

Property `Imag As Variant`

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

Example

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

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

```

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

```

Property MWFlags As MWFlags

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

Sub Clone(ppComplex As MWComplex)

Creates a copy of an MWComplex object.

Parameters

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

Return Value

None

Remarks

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

Class **MWSparse**

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

In this section...
“Property NumRows As Long” on page 17-30
“Property NumColumns As Long” on page 17-30
“Property RowIndex As Variant” on page 17-30
“Property ColumnIndex As Variant” on page 17-31
“Property Array As Variant” on page 17-31
“Property MWFlags As MWFlags” on page 17-31
“Sub Clone(ppSparse As MWSparse)” on page 17-31

Property **NumRows As Long**

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

Property **NumColumns As Long**

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

Property **RowIndex As Variant**

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

Property ColumnIndex As Variant

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

Property Array As Variant

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

Property MWFlags As MWFlags

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

Sub Clone(ppSparse As MWSparse)

Creates a copy of an **MWSparse** object.

Parameters

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

Return Value

None.

Remarks

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

Example

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

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

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

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

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

Class MWArg

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

In this section...

“Property Value As Variant” on page 17-34

“Property MWFlags As MWFlags” on page 17-34

“Sub Clone(ppArg As MWArg)” on page 17-34

Property Value As Variant

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

Property MWFlags As MWFlags

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

Sub Clone(ppArg As MWArg)

Creates a copy of an MWArg object.

Parameters

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

Return Value

None.

Remarks

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

Enum `mwArrayFormat`

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

`mwArrayFormat` Values

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

Enum mwDataType

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

mwDataType Values

Constant	Numeric Value	MATLAB Type
<code>mwTypeDefault</code>	0	Not applicable
<code>mwTypeLogical</code>	3	logical
<code>mwTypeChar</code>	4	char
<code>mwTypeDouble</code>	6	double
<code>mwTypeSingle</code>	7	single
<code>mwTypeInt8</code>	8	int8
<code>mwTypeUInt8</code>	9	uint8
<code>mwTypeInt16</code>	10	int16
<code>mwTypeUInt16</code>	11	uint16
<code>mwTypeInt32</code>	12	int32
<code>mwTypeUInt32</code>	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
<code>mwDateFormatNumeric</code>	0	Format dates as numeric values
<code>mwDateFormatString</code>	1	Format dates as strings

Deploying .NET Components With the F# Programming Language

The Magic Square Example Using F#

The F# programming language offers the opportunity to implement the same solutions you usually implement using C#, but with less code. This can be helpful when scaling a deployment solution across an enterprise-wide installation, or in any situation where code efficiency is valued. The brevity of F# programs can also make them easier to maintain.

The following example summarizes how to integrate the deployable MATLAB `magic` function from “Create a .NET Assembly From MATLAB Code” on page 1-11 in this user's guide.

You must be running Microsoft Visual Studio 2010 or higher to use this example.

For more information about the F# language, go to <http://fsharp.net>.

Prerequisites

If you build this example on a system running 64-bit Microsoft Visual Studio, you must add a reference to the 32-bit `MWArray` DLL due to a current imitation of Microsoft's F# compiler.

Step 1: Build the Component

Build the `makeSqr` component using the instructions in “Create a .NET Assembly From MATLAB Code” on page 1-11.

Step 2: Integrate the Component Into an F# Application

- 1 Using Microsoft Visual Studio 2010 or higher, create an F# project.
- 2 Add references to your component and `MWArray` in Visual Studio.
- 3 Make the .NET namespaces available for your component and `MWArray` libraries:

```
open makeSqr
open MathWorks.MATLAB.NET.Arrays
```

- 4 Define the Magic Square function with an initial `let` statement, as follows:

```
let magic n =
```

Then, add the following statements to complete the function definition.

- a** Instantiate the Magic Square component:

```
use magicComp = new makeSqr.MLTestClass()
```

- b** Define the input argument:

```
use inarg = new MWNumericArray((int) n)
```

- c** Call MATLAB, get the output argument cell array, and extract the first element as a two-dimensional float array:

```
(magicComp.makesquare(1, inarg).[0].ToArray() :?> float[,])
```

The complete function definition looks like this:

```
let magic n =
    // Instantiate the magic square component
    use magicComp = new makeSqr.MLTestClass()
    // Define the input argument
    use inarg = new MWNumericArray((int) n)
    // Call MATLAB, get the output argument cell array,
    // extract the first element as a 2D float array
    (magicComp.makesquare(1, inarg).[0].ToArray()
     :?> float[,])
```

- 5** Add another let statement to define the output display logic:

```
let printMagic n =
    let numArray = magic n
    // Display the output
    printfn "Number of [rows,cols]: [%d,%d]"
        (numArray.GetLength(0)) (numArray.GetLength(1))
    printfn ""
    for i in 0 .. numArray.GetLength(0)-1 do
        for j in 0 .. numArray.GetLength(1)-1 do
            printf "%3.0f " numArray.[i,j]
        printfn ""
    printfn "=====\n"

ignore(List.iter printMagic [1..19])
// Pause until keypress
ignore(System.Console.ReadKey())
```

The complete program listing follows:

The F# Magic Square Program

```
open makeSqr
open MathWorks.MATLAB.NET.Arrays
let magic n =
    // Instantiate the magic square component
    use magicComp = new makeSqr.MLTestClass()
    // Define the input argument
    use inarg = new MWNumericArray((int) n)
    // Call MATLAB, get the output argument cell array,
    // extract the first element as a 2D float array
    (magicComp.makesquare(1, inarg).[0].ToArray() :?> float[,])

let printMagic n =
    let numArray = magic n
    // Display the output
    printfn "Number of [rows,cols]: [%d,%d]"
        (numArray.GetLength(0)) (numArray.GetLength(1))
    printfn ""
    for i in 0 .. numArray.GetLength(0)-1 do
        for j in 0 .. numArray.GetLength(1)-1 do
            printf "%3.0f " numArray.[i,j]
        printfn ""
    printfn "=====\n"

ignore(List.iter printMagic [1..19])
// Pause until keypress
ignore(System.Console.ReadKey())
```

Step 3: Deploy the Component

See “Distributing MATLAB Code Using the MATLAB Runtime” on page 6-2 for information about deploying your component to end users.