MATLAB® Compiler SDK™

.NET User's Guide

MATLAB®

MathWorks®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

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

MATLAB® Compiler SDK™ .NET User's Guide

© COPYRIGHT 2002–2016 by 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 2015	Online only	New for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)

Contents

Getting Started

1

Component Integration

2

C# Integration Examples

3

Microsoft Visual Basic Integration Examples

4

Distribute Integrated .NET Applications

5

Distribute to End Users

6

Type-Safe Interfaces, WCF, and MEF

7

Web Deployment of Figures and Images

8

Windows Communications Foundation Based Components

9

.NET Remoting

10

Troubleshooting

11

Reference Information

12

13

Deploying .NET Components With the F# Programming Language

A

Getting Started

MATLAB Compiler SDK .NET Target Requirements

In this section...
"Microsoft .NET Framework Installation" on page 1-2
"MATLAB Compiler SDK .NET Limitations" on page 1-2
"Supported Microsoft .NET Framework Versions" on page 1-2
"System and Product Requirements" on page 1-3
"Path Modifications Required for Accessibility" on page 1-3

Microsoft .NET Framework Installation

Install the supported version of the Microsoft® .NET Framework. Your ability to use the latest MATLAB Compiler SDK functionality often depends on having the most current version of the framework installed. For a list of supported Framework versions, see "Supported Microsoft .NET Framework Versions" on page 1-2.

MATLAB Compiler SDK .NET Limitations

Using `addAssembly` (External Interfaces)

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

Serialization of MATLAB Objects Unsupported

There is no support in MATLAB Compiler SDK for serializing MATLAB objects from MATLAB into .NET code.

Supported Microsoft .NET Framework Versions

MATLAB Compiler SDK supports the following versions of Microsoft .NET Framework.

- 2.0
- 3.0
- 3.5
- 4.0 (Managed Extensibility Framework feature only)

- 4.5

System and Product Requirements

You must have the MATLAB and MATLAB Compiler™ products installed to install the MATLAB Compiler SDK product.

The MATLAB Compiler SDK .NET target is available only on Windows®.

For an up-to-date list of all the system and compiler software supported by MATLAB, MATLAB Compiler, and MATLAB Compiler SDK 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:

```
matlabroot\sys\java\jre\arch\jre\bin\JavaAccessBridge.dll
matlabroot\sys\java\jre\arch\jre\bin\WindowsAccessBridge.dll
```
You may not be able to use such technologies without doing so.

2

Component Integration

Common Integration Tasks and Naming Conventions

In this section...
"Component Access On Another Computer" on page 2-2
"Component and Class Naming Conventions" on page 2-3

In "Integrate a .NET Assembly Into a C# Application" on page 2-4, 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.

Component Access On Another Computer

To implement your .NET assembly 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".

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

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

   You must also add a reference to the `MWArray` component in *matlabroot*\toolbox\dotnetbuilder\bin*architecture**framework_version*. See "Supported Microsoft .NET Framework Versions" on page 1-2 for a list of supported framework versions.

3 Instantiate the generated .NET 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

---

To avoid using data conversion classes, see "Implement a Type-Safe Interface" on page 7-10.

Component and Class Naming Conventions

Typically you should specify names for assemblies and classes that will be clear to programmers who use the generated code. 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 Compiler SDK .NET examples use Pascal case.

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

Integrate a .NET Assembly Into a C# Application

This example shows how to call a .NET assembly from a C# application. To create the .NET assembly from your MATLAB function, see "Create a .NET Assembly".

1 Install the .NET assembly and the MATLAB Runtime.

The assembly and the MATLAB Runtime are both installed by the Library Compiler app. 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 Copy the following C# code into the project and save it.

```csharp
// Make .NET Namespaces available to your generated component.
using System;
using System.Collection.Generic;
using System.Text;

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

// 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;

    // Because class instantiation and method invocation make their exceptions at
    // you should enclose your code in a try-catch block to handle errors.
    try
    {
      // Instantiate your component class.
      obj = new MLTestClass();

      // Invoke your component.
      input = 5;
      result = obj.makeSqr(1, input);

      // Extract the Magic Square you created from the first index of result
      output = (MWNumericArray)result[0];

      // print the output.
      Console.WriteLine(output);
    }
    catch
    {
      throw;
    }
  }
}
}
```

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

Note: 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);
```

Running the Component Installer

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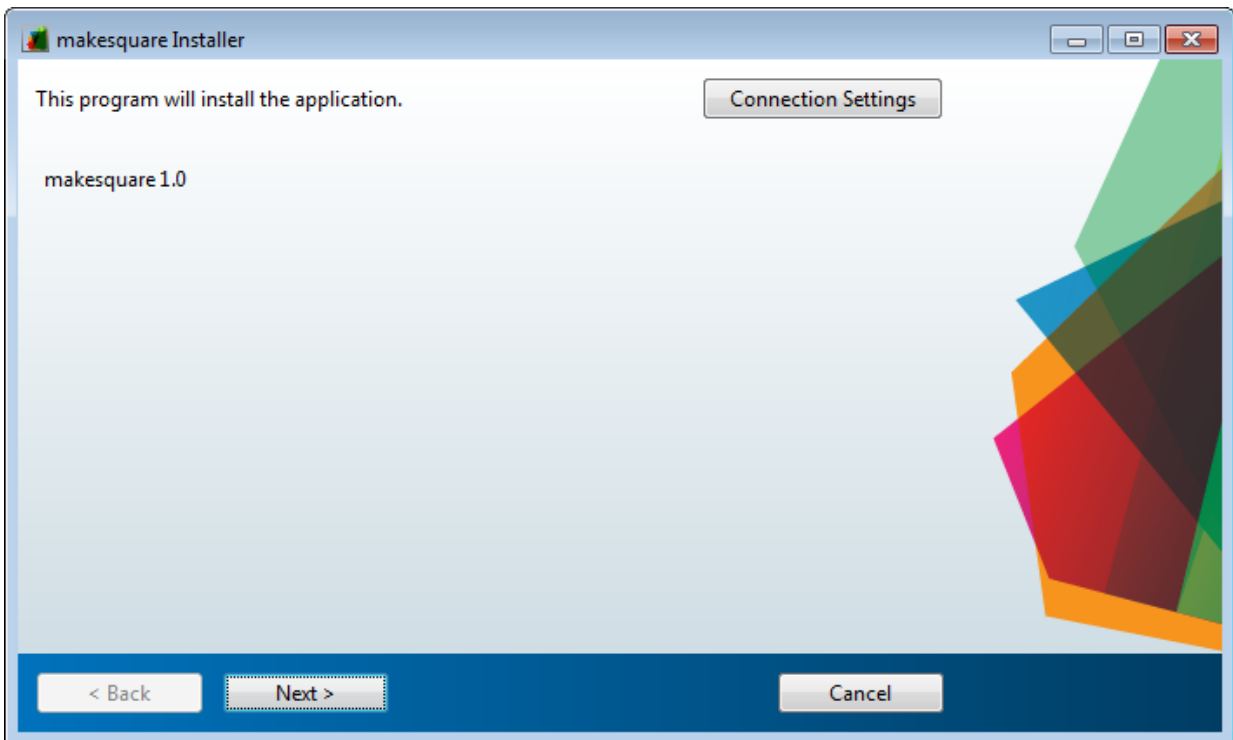
By clicking **Next** on each screen, the installer leads you through the installation process. During installation, you can specify the installation folder. The installer also automatically downloads the MATLAB Runtime, if needed.

Data Conversion Between .NET and MATLAB

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 "Rules for Data Conversion Between .NET and MATLAB" on page 12-2 for a complete list of rules to convert between .NET and MATLAB Compiler SDK data types.

Managing Data Conversion Issues with MATLAB Compiler SDK .NET Data Conversion Classes

To support data conversion between managed types and MATLAB types, MATLAB Compiler SDK 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 MATLAB Compiler SDK.

Overview of Classes and Methods in the Data Conversion Class Hierarchy

To support MATLAB data types, the MATLAB Compiler SDK product provides the `MWArray` data conversion classes in the `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

---

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

Automatic Casting to MATLAB Types

Note: Because the conversion process is typically automatic, you do not need to understand the conversion process to pass and return arguments with MATLAB Compiler SDK .NET assemblies.

In most instances, if a native .NET primitive or array is used as an input parameter in a C# program, the MATLAB Compiler SDK product transparently converts it to an instance of the appropriate `MWArray` class before it is passed on to the generated method. The MATLAB Compiler SDK product 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 MATLAB Compiler SDK product casts this argument to a MATLAB 1-by-1 double `MWNumericArray` type (which is a wrapper class containing a MATLAB double array).

See "Rules for Data Conversion Between .NET and MATLAB" on page 12-2 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

The MATLAB Compiler SDK product 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 .NET 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, use the MWNumericArray constructor that provides the ability to control the automatic conversion.

```
MWNumericArray array = new MWNumericArray(data, false);
```

The MATLAB Compiler SDK product does not support some MATLAB array types because they are not CLS-compliant. See "Unsupported MATLAB Array Types" on page 12-8 for a list of the unsupported types.

For more information about the concepts involved in data conversion, see "Managing Data Conversion Issues with MATLAB Compiler SDK .NET Data Conversion Classes" on page 2-8.

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 Compiler SDK 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= {O}", sum);
   sum= mySumClass.mysum((double)2, 4, 6, 8);
   Console.WriteLine("Sum= {O}", 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 "Rules for Data Conversion Between .NET and MATLAB" on page 12-2 for a discussion of the default data conversion for all supported .NET types).

Use the feval Interface

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

```
MyPrimesClassmyClass = new MyPrimesClass();
MWArray[] maxPrimes = new MWArray[1];
maxPrimes[O] = 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 `ith` vector is equal to `i`. The MATLAB Compiler SDK product generates a .NET interface to this function as follows:

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

Usage Example

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

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

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 `MWArray` 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)
{
 MWArray primes= null;
```

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

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

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

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

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.

Real or Imaginary Components Within Complex Arrays

In this section...
"Component Extraction" on page 2-20
"Returning Values Using Component Indexing" on page 2-20
"Assigning Values with Component Indexing" on page 2-21
"Converting MATLAB Arrays to .NET Arrays Using Component Indexing" on page 2-21

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);
```

Block Console Display When Creating Figures

In this section...
"WaitForFiguresToDie Method" on page 2-23
"Using WaitForFiguresToDie to Block Execution" on page 2-23

WaitForFiguresToDie Method

The MATLAB Compiler SDK 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 assembly created by the MATLAB Compiler SDK product.
- 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 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 assembly created by the MATLAB Compiler SDK 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 Compiler SDK to create a .NET assembly with the following properties:

Assembly 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 and Resources Management

When creating the .NET application it is a good practice to properly handle run-time errors and manage resources.

Error Handling

As with managed code, any errors that occur during execution of a 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 Compiler SDK 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: {O}", ex);
   return new double[O];
  }
}
```

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;
     }
}
```

Freeing Resources Explicitly

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`.

Object Passing by Reference

In this section...
"MATLAB Array" on page 2-29
"Wrappering and Passing .NET Objects with MWObjectArray" on page 2-29

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

---

Wrappering 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 Compiler SDK .NET Assembly

To pass an object into a MATLAB Compiler SDK assembly:

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.addItem(MWchar2Ascii,'C', 67);
```

Returning a Custom .NET Object in a MATLAB Function Using a Deployed MATLAB Compiler SDK .NET Assembly

You can also use `MWObjectArray` to clone an object inside a MATLAB Compiler SDK .NET Assembly. Continuing with the example in "Passing a .NET Object into a MATLAB Compiler SDK .NET Assembly" on page 2-29, 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 "Integrating MATLAB Optimization Routines with Objective Functions in .NET" on page 3-26 and the "Integrating MATLAB Optimization Routines with Objective Functions in .NET" on page 4-24.

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.

Use Multiple Assemblies in Single Application

In this section...
"Work with MATLAB Function Handles" on page 2-32
"Work with Objects" on page 2-34

When developing applications that use multiple MATLAB .NET assemblies, consider that the following cannot be shared between assemblies:

- MATLAB function handles
- MATLAB figure handles
- MATLAB objects
- C, Java®, and .NET objects
- Executable data stored in cell arrays and structures

Work with MATLAB Function Handles

MATLAB function handles can be passed between an application and the MATLAB Runtime instance from which it originated. However, a MATLAB function handle cannot be passed into a MATLAB Runtime instance other than the one in which it originated. For example, suppose you had two MATLAB functions, `get_plot_handle` and `plot_xy`, and `plot_xy` used the function handle created by `get_plot_handle`.

```
% Saved as get_plot_handle.m
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end

% Saved as plot_xy.m
function plot_xy(x, y, h)
h(x, y);
end
```

If you compiled them into two shared libraries, the call to `plot_xy` would throw an exception.

```csharp
using System;

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

using get_plot_handle;
using plot_xy;

namespace MathWorks.Examples.PlotApp
 {
  class PlotCSApp
    {
      static void Main(string[] args)
        {
          try
            {
              // Create objects for the generated functions
              get_plot_handle.Class1 plotter=
                new get_plot_handle.Class1();
              plot_xy.Class1 plot = new plot_xy.Class1();

              MWArray h = plotter.get_plot_handle('--rs', (double)2,
                'k','g', (double)10);

              double[] x_data = {1,2,3,4,5,6,7,8,9};
              double[] y_data = {2,6,12,20,30,42,56,72,90};
              MWArray x = new MWArray(x_data);
              MWArray y = new MWArray(y_data);
              plot.plot_xy(x, y, h);
            }

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

The correct way to handle the situation is to compile both functions into a single assembly.

```
using System;

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

using plot_functions;

namespace MathWorks.Examples.PlotApp
 {
  class PlotCSApp
    {
      static void Main(string[] args)
        {
          try
            {
              // Create object for the generated functions
              Class1 plot= new Class1();

              MWArray h = plot.get_plot_handle('--rs', (double)2,
                'k','g', (double)10);

              double[] x_data = {1,2,3,4,5,6,7,8,9};
              double[] y_data = {2,6,12,20,30,42,56,72,90};
              MWArray x = new MWArray(x_data);
              MWArray y = new MWArray(y_data);
              plot.plot_xy(x, y, h);
            }

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

Work with Objects

MATLAB Compiler SDK enables you to return the following types of objects from the
MATLAB Runtime to your application code:

• MATLAB

- C++
- .NET
- Java

However, you cannot pass an object created in one MATLAB Runtime instance into a different MATLAB Runtime instance. This conflict can happen when a function that returns an object and a function that manipulates that object are compiled into different assemblies.

For example, you develop two functions. The first creates a bank account for a customer based on some set of conditions. The second transfers funds between two accounts.

```matlab
% Saved as account.m
classdef account < handle

    properties
        name
    end

    properties (SetAccess = protected)
        balance = 0
        number
    end

    methods
        function obj = account(name)
            obj.name = name;
            obj.number = round(rand * 1000);
        end

        function deposit(obj, deposit)
            new_bal = obj.balance + deposit;
            obj.balance = new_bal;
        end

        function withdraw(obj, withdrawl)
            new_bal = obj.balance - withdrawl;
            obj.balance = new_bal;
        end

    end
end

% Saved as open_acct .m
```

```
function acct = open_acct(name, open_bal )

    acct = account(name);

    if open_bal > 0
        acct.deposit(open_bal);
    end

end

% Saved as transfer.m
function transfer(source, dest, amount)

    if (source.balance > amount)
        dest.deposit(amount);
        source.withdraw(amount);
    end

end
```

If you compiled `open_acct.m` and `transfer.m` into separate assemblies, you could not transfer funds using accounts created with `open_acct`. The call to `transfer` throws an exception. One way of resolving this is to compile both functions into a single assembly. You could also refactor the application such that you are not passing MATLAB objects to the functions.

3

C# Integration Examples

Simple Plot

In this section...
"Purpose" on page 3-2
"Procedure" on page 3-2

Purpose

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

- Use the MATLAB Compiler SDK product to convert a MATLAB function (`drawgraph`) to a method of a .NET class (`Plotter`) and wrap the class in a .NET assembly (`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

---

- 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\VS`*Version*`\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" 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\VS*Version*\PlotExample \PlotCSApp\PlotApp.cs.

The program listing is shown here.

```csharp
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.`

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.

Variable Number of Arguments

Note: This example is similar to "Simple Plot" on page 3-2, except that the MATLAB function to be encapsulated takes a variable number of arguments instead of just one.

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

- How to use the MATLAB Compiler SDK product 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 assembly (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

---

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

Procedure 3.2. 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\VS*Version*\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)

    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" 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

```
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`.

    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.

Multiple Classes in a .NET Component

In this section...
"Purpose" on page 3-10
"Procedure" on page 3-11

Purpose

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

- How to use the MATLAB Compiler SDK product to create an assembly (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

---

- 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\VS*Version*\NET\SpectraExample\SpectraComp

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 PlotFFT(fftData, freq, powerSpect)

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\VS*Version*\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" 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

```csharp
using System;

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

using SpectraComp;

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

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

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

              Random random= new Random();

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

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

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

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

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

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

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

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

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

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

      for (int idx= 1; idx <= numElements; idx++)
        {
          resultArray[idx]= ((MWNumericArray)argsOut[2])[idx];
        }

      Console.WriteLine("Power Spectral Density:\n{0}", resultArray);

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

      // Plot the fft and power spectral density for the data array
      plotter.plotfft(argsOut[0], argsOut[1], argsOut[2]);

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

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

   #endregion
  }
}
```

The program does the following:

- Constructs an input array with values representing a random signal with two sinusoids at 15 and 40 Hz embedded inside of it
- Creates an MWNumericArray array that contains the data

- Instantiates a `SignalAnalyzer` object
- Calls the `computefft` method, which computes the FFT, frequency, and the spectral density
- Instantiates a `Plotter` object
- Calls the `plotfft` method, which plots the data
- Uses a `try/catch` block to handle exceptions

The following statement

```
MWNumericArray data= new MWNumericArray(MWArrayComplexity.Real,
MWNumericType.Double, numSamples);
```

shows how to use the `MWArray` class library to construct a `MWNumericArray` that is used as method input to the `computefft` function.

The following statement

```
SignalAnalyzer signalAnalyzer = new SignalAnalyzer();
```

creates an instance of the class `SignalAnalyzer`, and the following statement

```
MWArray[] argsOut= signalAnalyzer.computefft(3, data, interval);
```

calls the method `computefft`.

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

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

   b Add a reference to the `MWArray` component, which is *matlabroot*\toolbox \dotnetbuilder\bin*architecture**framework_version* \mwarray.dll. See "Supported Microsoft .NET Framework Versions" on page 1-2 for a list of supported framework versions.

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

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

Multiple MATLAB Functions in a Component Class

In this section...

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

---

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

This example builds a .NET component to perform matrix math. The example creates a program that performs Cholesky, LU, and QR factorizations on a simple tridiagonal matrix (finite difference matrix) with the following form:

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

You supply the size of the matrix on the command line, and the program constructs the matrix and performs the three factorizations. The original matrix and the results are printed to standard output. You may optionally perform the calculations using a sparse matrix by specifying the string "sparse" as the second parameter on the command line.

3-15

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\VS*Version*\NET\MatrixMathExample

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

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

    The code for the `cholesky`, `ludecomp`, and `qrdecomp` functions is already in your work folder in `MatrixMathExample\MatrixMathComp\`.

3 From the MATLAB apps gallery, open the **Library Compiler** app.

4 Build the .NET component. See the instructions in "Create a .NET Assembly" for more details. Use the following information:

Project Name	`MatrixMathComp`
Class Name	`Factor`
Files to compile	`cholesky ludecomp qrdecomp`

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

```
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 3-20 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* `\mwarray.dll.` See "Supported Microsoft .NET Framework Versions" on page 1-2 for a list of supported framework versions.

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

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

MATLAB Functions to Be Encapsulated

The following code defines the MATLAB functions used in the example.

cholesky.m

```
function [L] = Cholesky(A)

    L = chol(A);
```

ludecomp.m

```
function [L,U] = LUDecomp(A)

    [L,U] = lu(A);
```

qrdecomp.m

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

    [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

In this section...

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

---

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" 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\VS`*Version*`\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

```csharp
/* 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;       /* Output 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 a MATLAB function that determines external phone number */
    result= thePhonebook.makephone(1, friends);
    book= (MWStructArray)result[0];

    Console.WriteLine("Result: ");
    Console.WriteLine(book.ToString());

    /* Extract some data from the returned structure */
    Console.WriteLine("Result record 2:");

    Console.WriteLine(book["name", 2]);
    Console.WriteLine(book["phone", 2]);
    Console.WriteLine(book["external", 2]);

    /* Print the entire result structure using the helper function below */
    Console.WriteLine("");
    Console.WriteLine("Entire structure:");

    DispStruct(book);

    Console.ReadLine();
  }

public static void DispStruct(MWStructArray arr)
  {
    Console.WriteLine("Number of Elements: " + arr.NumberOfElements);

    int[] dims= arr.Dimensions;

    Console.Write("Dimensions: " + dims[0]);

    for (int idx= 1; idx < dims.Length; idx++)
      {
        Console.WriteLine("-by-" + dims[idx]);
      }

    Console.WriteLine("\nNumber of Fields: " + arr.NumberOfFields);
    Console.WriteLine("Standard MATLAB view:");
    Console.WriteLine(arr.ToString());

    Console.WriteLine("Walking structure:");

    string[] fieldNames= arr.FieldNames;

    for (int element= 1; element <= arr.NumberOfElements; element++)
      {
        Console.WriteLine("Element " + element);

        for (int field= 0; field < arr.NumberOfFields; field++)
          {
```

```
                     MWArray fieldVal= arr[arr.FieldNames[field], element];

                     /* Recursively print substructures, */
                     /*  give string display of other classes */
                     if (fieldVal.GetType() == typeof(MWStructArray))
                       {
                         Console.WriteLine("   " + fieldNames[field] + ":
                             nested structure:");
                         Console.WriteLine("+++ Begin of \"" + fieldNames[field] + "\"
                             nested structure");
                         DispStruct((MWStructArray)fieldVal);
                         Console.WriteLine("+++ End of \"" + fieldNames[field] +
                             "\" nested structure");
                       }

                     else
                       {
                         Console.Write("   " + fieldNames[field] + ": ");
                         Console.WriteLine(fieldVal.ToString());
                       }
                   }
               }
           }
       }
```

The program does the following:

- Creates a structure array, using MWStructArray to represent the example phonebook data.

- Instantiates the Phonebook class as thePhonebook object, as shown:
  thePhonebook = new phonebook();

- Calls the makephone method to create a modified copy of the structure by adding an additional field, as shown:
  result = thePhonebook.makephone(1, friends);

6 Build thePhoneBookCSApp application using Visual Studio .NET.

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

   b Add a reference to the MWArray component, which is *matlabroot*\toolbox \dotnetbuilder\bin*architecture**framework_version* \mwarray.dll.

   c If necessary, add (or fix the location of) a reference to the PhoneBookComp component which you built in a previous step. (The component, PhoneBookComp.dll, is in the \PhoneBookExample\PhoneBookComp \x86\V2.0\Debug\distrib subfolder of your work area.)

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

The `PhoneBookApp` program should display the output:

```
Friends:
2x2 struct array with fields:
    name
    phone
Result:
2x2 struct array with fields:
    name
    phone
    external
Result record 2:
Mary Smith
3912
(508) 555-3912

Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
Walking structure:
Element 1
   name: Jordan Robert
   phone: 3386
   external: (508) 555-3386
Element 2
   name: Mary Smith
   phone: 3912
   external: (508) 555-3912
Element 3
   name: Stacy Flora
   phone: 3238
   external: (508) 555-3238
Element 4
   name: Harry Alpert
   phone: 3077
   external: (508) 555-3077
```

Integrating MATLAB Optimization Routines with Objective Functions in .NET

In this section...
"Purpose" on page 3-26
"OptimizeComp Component" on page 3-26
"Procedure" on page 3-27

Purpose

This example shows how to:

- Use the MATLAB Compiler SDK product to create an assembly (`OptimizeComp`). This assembly 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

- 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 \VS*Version*\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\VS*Version*\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.

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\VS*Version*\NET\Examples \OptimizeExample\OptimizeComp.

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

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

directEval = h.evaluateFunction(xO)
wrapperEval = mWrapper(xO)

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

    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\VS*Version*\NET\Examples\OptimizeExample \OptimizeCSApp. The program listing for `BananaFunction.cs` displays the following code:

```
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
```

4

Microsoft Visual Basic Integration Examples

Note: The examples for the MATLAB Compiler SDK product are in *matlabroot*\toolbox\dotnetbuilder\Examples\VS*Version*, where *matlabroot* is the folder where the MATLAB product is installed and VS*Version* 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\VS*Version*\DotNetExamples.sln

Magic Square

To create the component for this example, see the first several steps in "Create a .NET Assembly". 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\VS*Version*\NET\MagicSquareExample\MagicSquareVBApp

The program listing is as follows.

MagicSquareApp.vb

```vb
Imports System
Imports System.Reflection
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports MagicSquareComp

Namespace MathWorks.Examples.MagicSquare

    ' <summary>
    ' The MagicSquareApp class computes a magic square of the user specified size.
    ' </summary>
    ' <remarks>
    ' args[0] - a positive integer representing the array size.
    ' </remarks>
    Class MagicSquareApp

#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)

            Dim arraySize As MWNumericArray = Nothing
            Dim magicSquare As MWNumericArray = Nothing

            Try
                ' Get user specified command line arguments or set default
                If (0 <> args.Length) Then
                    arraySize = New MWNumericArray(Int32.Parse(args(0)), False)
                Else
                    arraySize = New MWNumericArray(4, False)
                End If

                ' Create the magic square object
                Dim magic As MagicSquareClass = New MagicSquareClass

                ' Compute the magic square and print the result
```

```
            magicSquare = magic.makesquare(arraySize)

            Console.WriteLine("Magic square of order {0}{1}{2}{3}", arraySize,
              Chr(10), Chr(10), magicSquare)

            ' Convert the magic square array to a two dimensional native double array
            Dim nativeArray(,) As Double =
              CType(magicSquare.ToArray(MWArrayComponent.Real), Double(,))

            Console.WriteLine("{0}Magic square as native array:{1}", Chr(10), Chr(10))

            ' Display the array elements:
            Dim index As Integer = arraySize.ToScalarInteger()

            For i As Integer = 0 To index - 1
                For j As Integer = 0 To index - 1
                    Console.WriteLine("Element({0},{1})= {2}", i, j, nativeArray(i, j))
                Next j
            Next i

            Console.ReadLine() 'Wait for user to exit application

        Catch exception As Exception

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

        End Try
      End Sub
#End Region

    End Class

End Namespace
```

The application you build from this source file does the following:

- Lets you pass a dimension for the magic square from the command line.

- Converts the dimension argument to a MATLAB integer scalar value.

- Declares variables of type MWNumericArray to handle data required by the encapsulated makesquare function.

> **Note:** For 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

- 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.

4-3

- Displays the array elements on the command line.

2 Build the application using Visual Studio .NET.

    a The `MagicSquareVBApp` folder contains a Visual Studio .NET project file for each example. Open the project in Visual Studio .NET for this example by double-clicking `MagicSquareVBApp.vbproj` in Windows Explorer.

    b 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.

Simple Plot

To create the component for this example, see "Simple Plot" on page 3-2. Then create a Visual Basic application as follows:

1. Review the sample application for this example in *matlabroot*\toolbox \dotnetbuilder\Examples*VSVersion*\NET\PlotExample\PlotVBApp \PlotApp.vb.

   The program listing is shown here.

PlotApp.vb

```vb
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: {O}", 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 MWArray Class Library Reference*, available in the *matlabroot*\help \dotnetbuilder\MWArrayAPI folder, where *matlabroot* represents your MATLAB installation folder

---

- Uses a `try-catch` block to catch and handle any exceptions

The statement

```
Dim plotter As Plotter = New Plotter
```

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

```
plotter.drawgraph(coords)
```

calls the method `drawgraph`.

2 Build the `PlotApp` application using Visual Studio .NET.

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

    b Add a reference to the `MWArray` component, which is *matlabroot*\toolbox \dotnetbuilder\bin*architecture**framework_version*\mwarray.dll.

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

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

Variable Number of Arguments

To create the component for this example, see "Variable Number of Arguments" on page 3-6. Then create a Microsoft Visual Basic application as follows:

1 Review the sample application for this example in *matlabroot*\toolbox \dotnetbuilder\Examples\VS*Version*\NET\VarArgExample\VarArgVBApp \VarArgApp.vb.

The program listing is shown here.

VarArgApp.vb

```vb
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

---

- Uses a `try-catch-finally` block to catch and handle any exceptions

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

```
data = CType(plotter.drawgraph(colorSpec, coords(0), coords(1), coords(2),
       coords(3), coords(4)), MWNumericArray)
...
data = CType(plotter.drawgraph(colorSpec, coords), MWNumericArray)
```

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

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

        b   Add a reference to the `MWArray` component, which is *matlabroot*`\toolbox` `\dotnetbuilder\bin\`*architecture*`\`*framework_version*`\mwarray.dll`.

        c   If necessary, add (or update the location of) a reference to the `VarArgComp` component which you built in a previous step. (The component, `VarArgComp.dll`, is in the `\VarArgExample\VarArgComp\x86\V2.0\Debug` `\distrib` subfolder of your work area.)

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

Multiple Classes in a .NET Component

To create the component for this example, see the first few steps of the "Multiple Classes in a .NET Component" on page 3-10. Then create a Microsoft Visual Basic application as follows:

1 Review the sample application for this example in *matlabroot*\toolbox \dotnetbuilder\Examples\VS*Version*\NET\SpectraExample\SpectraVBApp \SpectraApp.vb.

The program listing is shown here.

SpectraApp.vb

```vb
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
```

4-11

```vb
                    ' Create a new signal analyzer object
                    Dim signalAnalyzer As SignalAnalyzer = New SignalAnalyzer

                    ' Compute the fft and power spectral density for the data array
                    Dim argsOut() As MWArray = signalAnalyzer.computefft(3, data,
                                               MWArray.op_Implicit(interval))

                    ' Print the first twenty elements of each result array
                    Dim numElements As Integer = 20
                    Dim resultArray As MWNumericArray =
                            New MWNumericArray(MWArrayComplexity.Complex,
                                    MWNumericType.Double, numElements)

                    For idx = 1 To numElements
                        resultArray(idx) = (CType(argsOut(0), MWNumericArray))(idx)
                    Next idx

                    Console.WriteLine("FFT:{0}{1}{2}", Chr(10), resultArray, Chr(10))

                    For idx = 1 To numElements
                        resultArray(idx) = (CType(argsOut(1), MWNumericArray))(idx)
                    Next idx

                    Console.WriteLine("Frequency:{0}{1}{2}", Chr(10), resultArray, Chr(10))

                    For idx = 1 To numElements
                        resultArray(idx) = (CType(argsOut(2), MWNumericArray))(idx)
                    Next idx

                    Console.WriteLine("Power Spectral Density:{0}{1}{2}",
                      Chr(10), resultArray, Chr(10))

                    ' Create a new plotter object
                    Dim plotter As Plotter = New Plotter

                    ' Plot the fft and power spectral density for the data array
                    plotter.plotfft(argsOut(0), argsOut(1), argsOut(2))

                    Console.ReadLine()  ' Wait for user to exit application

                Catch exception As Exception
                    Console.WriteLine("Error: {0}", exception)

                End Try
            End Sub
    #End Region

        End Class
    End Namespace
```

The program does the following:

- Constructs an input array with values representing a random signal with two sinusoids at 15 and 40 Hz embedded inside of it

- Uses `MWNumericArray` to handle data conversion

Note: For 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

---

- Instantiates a `SignalAnalyzer` object
- Calls the `computefft` method, which computes the FFT, frequency, and the spectral density
- Instantiates a `Plotter` object
- Calls the `plotfft` method, which plots the data
- Uses a `try/catch` block to handle exceptions

The following statements

```
Dim data As MWNumericArray = New MWNumericArray_
    (MWArrayComplexity.Real, MWNumericType.Double, numSamples)
...
Dim resultArray As MWNumericArray = New MWNumericArray_
    (MWArrayComplexity.Complex,
     MWNumericType.Double, numElements)
```

show how to use the `MWArray` class library to construct the necessary data types.

The following statement

```
Dim signalAnalyzer As SignalAnalyzer = New SignalAnalyzer
```

creates an instance of the class `SignalAnalyzer`, and the following statement

```
Dim argsOut() As MWArray =
    signalAnalyzer.computefft(3, data,
        MWArray.op_Implicit(interval))
```

calls the method `computefft` and request three outputs.

2 Build the `SpectraApp` application using Visual Studio .NET.

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

    b   Add a reference to the `MWArray` component, which is *matlabroot*`\toolbox` `\dotnetbuilder\bin\`*architecture*`\`*framework_version*`\mwarray.dll.`

    c   If necessary, add (or update the location of) a reference to the `SpectraComp` component which you built in a previous step. (The component, `SpectraComp.dll`, is in the `\SpectraExample\SpectraComp` `\x86\V2.0\Debug\distrib` subfolder of your work area.)

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

Multiple MATLAB Functions in a Component Class

To create the component for this example, see the first few steps in "Multiple MATLAB Functions in a Component Class" on page 3-15. Then create a Microsoft Visual Basic application as follows.

1 Review the sample application for this example in:

matlabroot\toolbox\dotnetbuilder\Examples\VS*Version*\NET\MatrixMathExample\MatrixMathVBApp\MatrixMathApp

The program listing is shown here.

MatrixMathApp.vb

```vb
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[O] - 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
```

```vb
    'Convert matrix order
    matrixOrder = Int32.Parse(args(0))

    If (0 > matrixOrder) Then
        Throw New ArgumentOutOfRangeException("matrixOrder", matrixOrder, _
         "Must enter a positive integer for the matrix order(N)")
    End If

    makeSparse = ((1 < args.Length) AndAlso (args(1).Equals("sparse")))
End If

' Create the test matrix.  If the second argument
'   is "sparse", create a sparse matrix.
matrix = IIf(makeSparse, _
MWNumericArray.MakeSparse(matrixOrder, matrixOrder,
                          MWArrayComplexity.Real,
                          (matrixOrder + (2 * (matrixOrder - 1)))), _
New MWNumericArray(MWArrayComplexity.Real, MWNumericType.Double,
                                  matrixOrder, matrixOrder))

' Initialize the test matrix
For rowIdx As Integer = 1 To matrixOrder
    For colIdx As Integer = 1 To matrixOrder
        If rowIdx = colIdx Then
            matrix(rowIdx, colIdx) = New MWNumericArray(2.0)
        ElseIf colIdx = rowIdx + 1 Or colIdx = rowIdx - 1 Then
            matrix(rowIdx, colIdx) = New MWNumericArray(-1.0)
        End If
    Next colIdx
Next rowIdx

' Create a new factor object
Dim factor As Factor = New Factor

' Print the test matrix
Console.WriteLine("Test Matrix:{0}{1}{2}", Chr(10), matrix, Chr(10))

' Compute and print the cholesky factorization using
'    the single output syntax
argOut = factor.cholesky(matrix)

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

' Compute and print the LU factorization using the multiple output syntax
argsOut = factor.ludecomp(2, matrix)

Console.WriteLine("LU Factorization:
     {0}L Matrix:{1}{2}{3}U Matrix:{4}{5}{6}", Chr(10), Chr(10),
              argsOut(0), Chr(10), Chr(10), argsOut(1), Chr(10))

MWNumericArray.DisposeArray(argsOut)

' Compute and print the QR factorization
argsOut = factor.qrdecomp(2, matrix)

Console.WriteLine("QR Factorization:
     {0}Q Matrix:{1}{2}{3}R Matrix:{4}{5}{6}", Chr(10), Chr(10),
              argsOut(0), Chr(10), Chr(10), argsOut(1), Chr(10))

Console.ReadLine()
```

```
        Catch exception As Exception

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

        Finally

            ' Free native resources
            If Not (matrix Is Nothing) Then
                matrix.Dispose()
            End If
            If Not (argOut Is Nothing) Then
                argOut.Dispose()
            End If

            MWNumericArray.DisposeArray(argsOut)
        End Try
    End Sub
#End Region
    End Class
End Namespace
```

The statement

```
Dim factor As Factor = New Factor
```

creates an instance of the class `Factor`.

The following statements call the methods that encapsulate the MATLAB functions:

```
argOut = factor.cholesky(matrix)


argsOut = factor.ludecomp(2, matrix)

...
argsOut = factor.qrdecomp(2, matrix)
```

Note: See "Understanding the MatrixMath Program" on page 3-20 for more details about the structure of this program.

---

2 Build the `MatrixMathApp` application using Visual Studio .NET.

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

  b Add a reference to the `MWArray` component, which is *matlabroot*\toolbox \dotnetbuilder\bin*architecture**framework_version*\mwarray.dll.

4-17

    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

In this section...
"makephone Function" on page 4-19
"Procedure" on page 4-19

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\VS*Version*\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)

   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" 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\VS*Version*\NETPhoneBookExample \PhoneBookVBApp\PhoneBookApp.vb.

The program defines a structure array containing names and phone numbers, modifies it using a MATLAB function, and displays the resulting structure array.

The program listing is shown here.

PhoneBookApp.vb

```
' Necessary package imports

Imports MathWorks.MATLAB.NET.Arrays
Imports PhoneBookComp

'
' getphone class demonstrates the use of the MWStructArray class
'
Public Module PhoneBookVBApp
    Public Sub Main()
        Dim thePhonebook As phonebook    'Stores deployment class instance
        Dim friends As MWStructArray     'Sample input data
        Dim result As Object()           'Stores the result
        Dim book As MWStructArray        'Output 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())
```

```vb
        ' Pass it to a MATLAB function that determines external phone number
        result = thePhonebook.makephone(1, friends)
        book = CType(result(0), MWStructArray)
        Console.WriteLine("Result: ")
        Console.WriteLine(book.ToString())

        ' Extract some data from the returned structure '
        Console.WriteLine("Result record 2:")

        Console.WriteLine(book("name", 2))
        Console.WriteLine(book("phone", 2))
        Console.WriteLine(book("external", 2))

        ' Print the entire result structure using the helper function below
        Console.WriteLine("")
        Console.WriteLine("Entire structure:")
        dispStruct(book)
    End Sub


    Sub dispStruct(ByVal arr As MWStructArray)
        Console.WriteLine("Number of Elements: " + arr.NumberOfElements.ToString())
        'int numDims = arr.NumberofDimensions
        Dim dims As Integer() = arr.Dimensions
        Console.Write("Dimensions: " + dims(0).ToString())

        Dim i As Integer
        For i = 1 To dims.Length
            Console.WriteLine("-by-" + dims(i - 1).ToString())
        Next i
        Console.WriteLine("")
        Console.WriteLine("Number of Fields: " + arr.NumberOfFields.ToString())
        Console.WriteLine("Standard MATLAB view:")
        Console.WriteLine(arr.ToString())
        Console.WriteLine("Walking structure:")

        Dim fieldNames As String() = arr.FieldNames

        Dim element As Integer
        For element = 1 To arr.NumberOfElements
            Console.WriteLine("Element " + element.ToString())
            Dim field As Integer
            For field = 0 To arr.NumberOfFields - 1
                Dim fieldVal As MWArray = arr(arr.FieldNames(field), element)
                ' Recursively print substructures, give string display of other classes
                If (TypeOf fieldVal Is MWStructArray) Then
                    Console.WriteLine("   " + fieldNames(field) + ": nested structure:")
                    Console.WriteLine("+++ Begin of \"" + fieldNames[field] +
                                                    " \ " nested structure")

                    dispStruct(CType(fieldVal, MWStructArray))
                    Console.WriteLine("+++ End of \"" + fieldNames[field] +
                                                    " \ " nested structure")
                Else
                    Console.Write("   " + fieldNames(field) + ": ")
                    Console.WriteLine(fieldVal.ToString())
                End If
            Next field
        Next element
    End Sub
End Module
```

The program does the following:

- Creates a structure array, using `MWStructArray` to represent the example phonebook data.

- Instantiates the plotter class as `thePhonebook` object, as shown: `thePhonebook = new phonebook();`

- Calls the `makephone` method to create a modified copy of the structure by adding an additional field, as shown: `result = thePhonebook.makephone(1, friends);`

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

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

    b Add a reference to the `MWArray` component, which is *matlabroot*`\toolbox \dotnetbuilder\bin\`*architecture*`\`*framework_version*`\mwarray.dll`.

    c If necessary, add (or fix the location of) a reference to the `PhoneBookVBComp` component which you built in a previous step. (The component, `PhoneBookComp.dll`, is in the `\PhoneBookExample\PhoneBookVBApp \x86\V2.0\Debug\distrib` subfolder of your work area.)

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

The `getphone` program should display the output:

```
Friends:
2x2 struct array with fields:
    name
    phone
Result:
2x2 struct array with fields:
    name
    phone
    external
Result record 2:
Mary Smith
3912
(508) 555-3912
```

```
Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
Walking structure:
Element 1
    name: Jordan Robert
    phone: 3386
    external: (508) 555-3386
Element 2
    name: Mary Smith
    phone: 3912
    external: (508) 555-3912
Element 3
    name: Stacy Flora
    phone: 3238
    external: (508) 555-3238
Element 4
    name: Harry Alpert
    phone: 3077
    external: (508) 555-3077
```

Integrating MATLAB Optimization Routines with Objective Functions in .NET

Optimization Example

- "Purpose" on page 4-24
- "OptimizeComp Component" on page 4-24
- "Procedure" on page 4-25

Purpose

This example shows how to:

- Use the MATLAB Compiler SDK product to create an assembly (`OptimizeComp`). This assembly 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

- 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\VS*version*\NET\OptimizeExample\OptimizeVBApp.

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\VS*version*\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.

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\VS*Version*\NET
    \OptimizeExample\OptimizeComp.

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

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

directEval = h.evaluateFunction(xO)
wrapperEval = mWrapper(xO)

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

    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\VS*Version*\NET\OptimizeExample \OptimizeVBApp. The program listing for BananaFunction.vb displays the following code:

```
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 `OptimizeVBApp` 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
```

Distribute Integrated .NET Applications

Package .NET Applications

1. Gather and package the following files for installation on end user computers:

   - MATLAB Runtime installer

     See .
   - MATLAB generated .NET assembly
   - Executable for the application
2. Include directions for installing the MATLAB Runtime.

   See .

About the MATLAB Runtime

In this section...
"How is the MATLAB Runtime Different from MATLAB?" on page 5-3
"Performance Considerations and the MATLAB Runtime" on page 5-4

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

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

See "Install and Configure the MATLAB Runtime" on page 5-5 for more information.

How is the MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In the MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. The MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 6.3 (R2016b) of MATLAB Compiler, users who do not have MATLAB installed must have version 9.1 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MATLAB Runtime

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

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

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

Install and Configure the MATLAB Runtime

In this section...

Download the MATLAB Runtime Installer

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

Install the MATLAB Runtime Interactively

To install the MATLAB Runtime:

1   Start the MATLAB Runtime installer.

Computer	Steps
Windows	Double-click the compiled MATLAB code package self-extracting archive file, typically named *my_program*_pkg.exe, where *my_program* is the name of the MATLAB code. This extracts the MATLAB Runtime installer from the archive, along with all the files that make up the MATLAB Runtime. Once all the files have been extracted, the MATLAB Runtime installer starts automatically.
Linux®	Extract the contents of the compiled package, which is a Zip file on Linux systems, typically named, *my_program*_pkg.zip, where *my_program* is the name of the compiled MATLAB code. Use the unzip command to extract the files from the package. `unzip MCRInstaller.zip`
Mac	

Computer	Steps
	Run the MATLAB Runtime installer script, from the directory where you unzipped the package file, by entering: `./install` For example, if you unzipped the package and MATLAB Runtime installer in `\home\USER`, you run the `./install` from `\home\USER`. **Note:** On Mac systems, you may need to enter an administrator username and password after you run `./install`.

2 When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click **Next** to proceed with the installation.

3 Specify the folder in which you want to install the MATLAB Runtime in the **Folder Selection** dialog box.

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

4 Confirm your choices and click **Next**.

The MATLAB Runtime installer starts copying files into the installation folder.

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

6 Click **Finish** to exit the installer.

Install the MATLAB Runtime Non-Interactively

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

• silent—the installer runs as a background task and does not display any dialog boxes

- automated—the installer displays the dialog boxes but does not wait for user interaction

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

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

Running the Installer in Silent Mode

To install the MATLAB Runtime in silent mode:

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

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

---

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

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

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

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

3 View a log of the installation.

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

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

Customizing a Non-Interactive Installation

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

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

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

Multiple MATLAB Runtime Versions on Single Machine

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

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

MATLAB and MATLAB Runtime on Same Machine

You do not need to install MATLAB Runtime on your machine if your machine has MATLAB installed. The version of MATLAB should be the same as the version of MATLAB that was used to create the compiled MATLAB code. Also, to act as the MATLAB Runtime replacement, the MATLAB installation must include MATLAB Compiler.

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

Modifying the Path

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

- **Windows**

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

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

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

- **UNIX**

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

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

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

Uninstall MATLAB Runtime

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

Windows

1   Start the uninstaller.

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

    You can also launch the MATLAB Runtime uninstaller from the *mcr_root*\uninstall\bin*arch* folder, where *mcr_root* is your MATLAB Runtime installation folder and *arch* is an architecture-specific folder, such as `win64`.

2   Select the MATLAB Runtime from the list of products in the Uninstall Products dialog box and
3   Click **Next**.
4   Click **Finish**.

Linux

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

    ```
    rm -rf mcr_root
    ```

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

Mac

1 Exit the application.

2 Navigate to your MATLAB Runtime installation folder. For example, the installation folder might be named `MATLAB_Compiler_Runtime.app` in your Applications folder.

3 Drag your MATLAB Runtime installation folder to the trash, and then select **Empty Trash** from the Finder menu.

6

Distribute to End Users

Deploy Components to End Users

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 available for downloading from the web to simplify the distribution of your applications created using the MATLAB Compiler or the MATLAB Compiler SDK. Download the MATLAB Runtime from the MATLAB Runtime product page.

The MATLAB Runtime installer does the following:

1 Install the MATLAB Runtime.

2 Install the component assembly in the folder from which the installer is run.

3 Copy the MWArray assembly to the Global Assembly Cache (GAC), as part of installing the MATLAB Runtime.

MATLAB Runtime Prerequisites

1 The MATLAB Runtime installer requires administrator privileges to run.

2 The version of the MATLAB Runtime that runs your application on the target computer must be compatible with the version of MATLAB Compiler or MATLAB Compiler SDK that built the deployed 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 or MATLAB Compiler SDK 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 website.

- **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, modify the path on the target machine. Setting the paths enables your application to find the MATLAB Runtime. For more information on setting the path, see "MATLAB Runtime Path Settings for Run-Time Deployment".

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, it puts the `MWArray` assembly in *mcrRoot*`\toolbox\dotnetbuilder\bin` `\`*architecture*`\`*framework_version*.
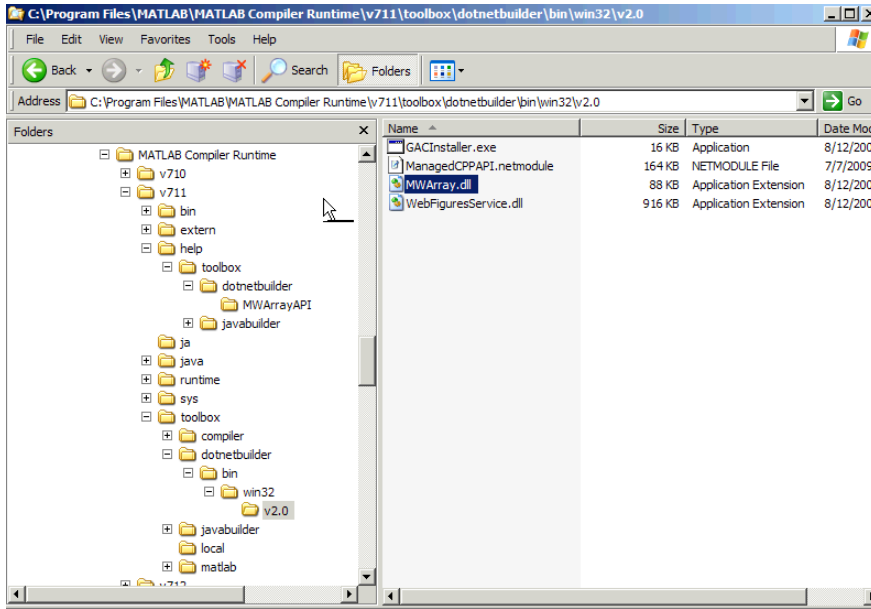
Sample Directory Structure of the MATLAB Runtime Including MWArray.dll

MATLAB Runtime Run-Time Options

In this section...
"What Run-Time Options Can You Specify?" on page 6-5
"Getting MATLAB Runtime Option Values Using MWMCR" on page 6-5

What Run-Time Options Can You Specify?

You can pass the options `-nojvm` and `-logfile` to MATLAB Compiler SDK from a .NET 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 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
`MWMCR.IsMCRInitialized()`	Returns `true` if the MATLAB Runtime run-time is initialized, otherwise returns `false`.
`MWMCR.IsMCRJVMEnabled()`	Returns `true` if the MATLAB Runtime run-time is launched with .NET Virtual Machine (JVM™), otherwise returns `false`.
`MWMCR.GetMCRLogFileName()`	Returns the name of the log file passed with the `LOGFILE` attribute.

Default 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)	`NOJVM(false)`
Log file usage	`LOGFILE(null)`

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 MATLAB Runtime with or without a JVM. It takes a Boolean as input. For example, NOJVM(true) launches the MATLAB Runtime without a JVM.
LOGFILE	Lets users pass the name of a log file, taking the file name as input. For example, LOGFILE("*logfile3.txt*") .

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);
               }
           }
       }
   }
```

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 instance, 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-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 MATLAB Runtime 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 Compiler SDK software supports a per-MATLAB Runtime instance state access through an object-oriented API. Access to a per-instance state is optional. 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 6-7.

For more information, see "The MATLAB Runtime User Data Interface".

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.

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.

```matlab
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(gcp);
disp('done');
speedup = (time1/time2);
```

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

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

3 Verify that you get the following results;

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 worke
Normal loop times: 0.7587,parallel loop time: 2.9988
parallel speedup:  0.253 times faster than normal
Parallel pool using the 'local' profile is shutting down.
done

a =

    0.2530
```

Step 2: Set the Parallel Computing Toolbox Profile

In order to compile MATLAB code to a .NET component and utilize 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 SDK.

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 app as follows:

1. Follow the steps in "Create a .NET Assembly" 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 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:

MATLAB Runtime Component Cache and Deployable Archive Embedding

Deployable archive data is automatically embedded directly in .NET Assemblies by default and extracted to a temporary folder.

Automatic embedding enables usage of the 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).
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

Environment Variable	Purpose	Notes
		`.max_size`, which resides in the file designated by running the `mcrcachedir` command, with the desired cache size limit.

Note: If you run `mcc` specifying conflicting wrapper and target types, the 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 archive embedded in it, as if you had specified a `-C` option to the command line.

Caution Do not extract the files within the `.ctf` file and place them individually under version control. Since the `.ctf` file contains interdependent MATLAB functions and data, the files within it must be accessed only by accessing the `.ctf` file. For best results, place the entire `.ctf` file under version control.

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())
```

6-15

```
        {
            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 Compiler SDK .NET assembly 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 `MWArray`

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).
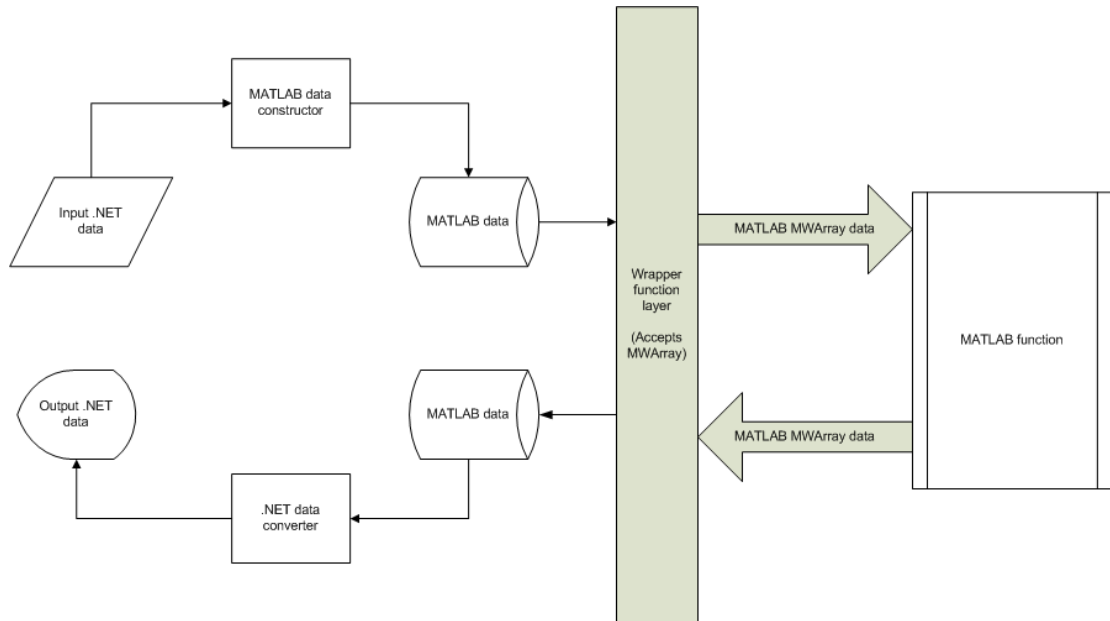
Manual Data Marshaling Without a Type-Safe Interface

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 Compiler SDK 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. For more information, see "Implement a Type-Safe Interface" on page 7-10.
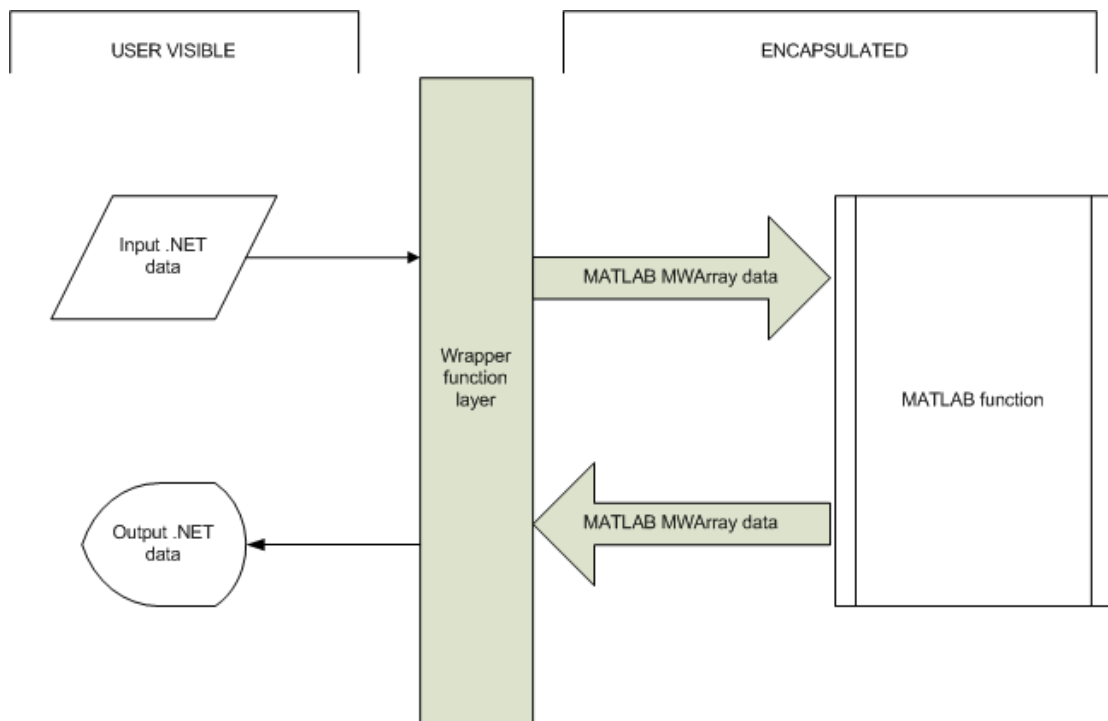
Simplified Data Marshaling With a Type-Safe Interface

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 the `MWArray` API.

- **You minimize cost of data you must marshal** by either placing `MWArray` objects in type-safe interfaces or by calling `MWArray`functions in the deployed MATLAB code.

- **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 Compiler SDK .NET assembly exports one or more public methods that accept and return data using `MWArray` objects.

Adding a type-safe interface to a MATLAB Compiler SDK assembly 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.
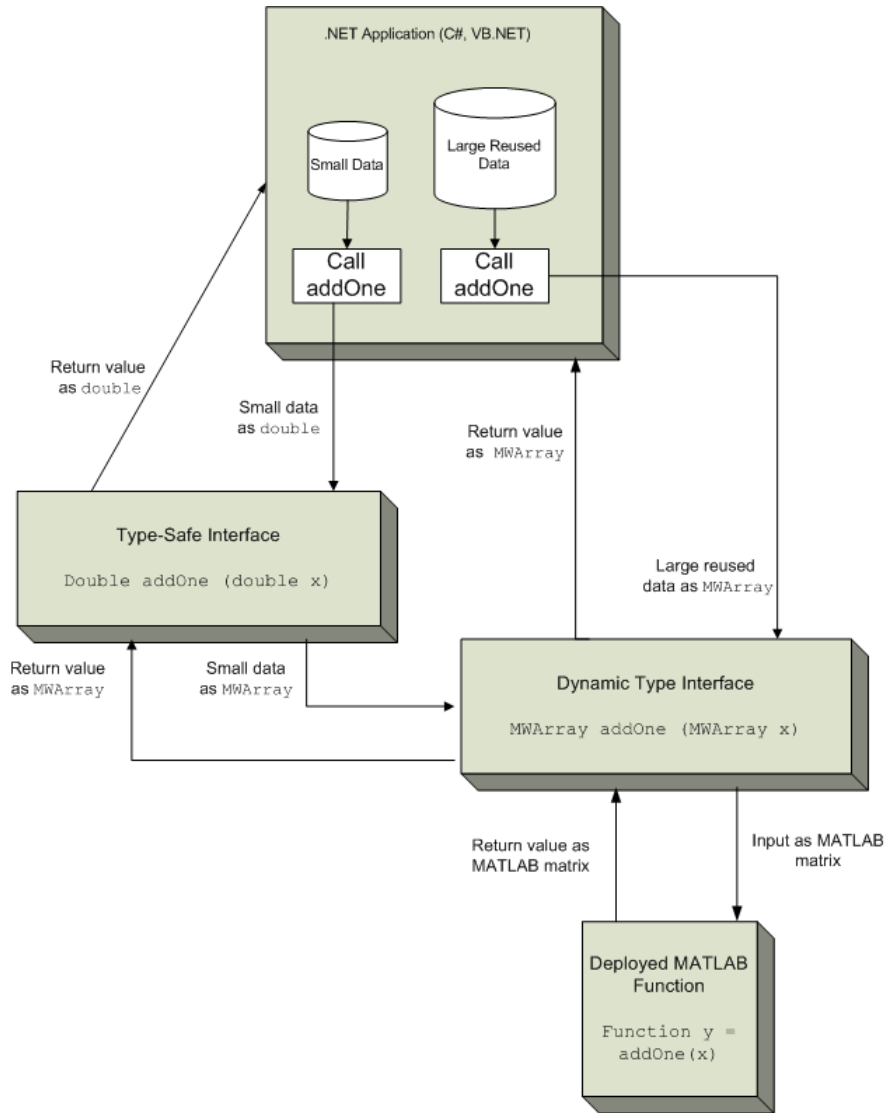
Architecture of a Deployed Component with a Type-Safe Interface

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` 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.

Generate the Type-Safe API with an Assembly

In this section...
"Use the Library Compiler App" on page 7-8
"Use the Command-Line Tools" on page 7-8

Use the Library Compiler App

The Library Compiler app generates the type-safe API, when you build your assembly, 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 app 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.

Use 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 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 .NET assembly to wrap with a type-safe API. If the assembly 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` for details on all command options.

Implement 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 Compiler SDK 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.
  - A function in the interface may have fewer inputs than its corresponding MATLAB function, 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 generated by MATLAB Compiler SDK.

See ".NET Types to MATLAB Types" on page 12-3 for complete guidelines in managing data conversion with type-safe interfaces.

Create Managed Extensibility Framework (MEF) Plug-Ins

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 at least Microsoft Visual Studio 2010 to create MEF applications. If you can't use Microsoft 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 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\VS*Version*\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 7-14
2. "Create a Contract Interface Assembly" on page 7-15
3. "Create a Metadata Attribute Assembly" on page 7-16
4. "Add Contract and Attributes References to MEFHost" on page 7-17
5. "Compile Your Code in Microsoft Visual Studio" on page 7-17
6. "Write MATLAB Functions for MEF Parts" on page 7-17
7. "Create Metadata Files" on page 7-18
8. "Build .NET Components from MATLAB Functions and Metadata" on page 7-18
9. "Install MEF Parts" on page 7-19
10. "Run the MEF Host Program" on page 7-20

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. Click **Assemblies > Framework** 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 run time.

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 Name the configuration file `App.config` and click **Add**.

e Replace the automatically-generated contents of `App.config` with this configuration:

```xml
<?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:

```csharp
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".

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 *component*Native.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 `mclmcrrt` 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).

8

Web Deployment of Figures and Images

Install WebFigureControl Into Microsoft Visual Studio Toolbox

1 If the toolbox is not displayed, open using **View > Toolbox**.
2 Select **Tools > Choose Toolbox Items**.
3 Click **Browse...**.
4 Select *matlabroot*\toolbox\dotnetbuilder\bin
arch\v2.0\WebFiguresService.dll.

If you are using the MATLAB Runtime, *matlabroot* is the location of the installed MATLAB Runtime.

If you are using MATLAB Compiler SDK, *matlabroot* is the location of the installed MATLAB Compiler SDK.
5 Click **OK**.

Once WebFiguresService.dll is added, you will see the following WebFigureControl in the **General** section of the Microsoft Visual Studio toolbar:
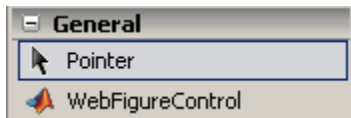
Quick Start Implementation of WebFigures

In this section...
"Assumptions About the Example" on page 8-3
"Procedure" on page 8-3

Assumptions About the Example

To work with this example:

- Assume the following MATLAB function has been created:

```
function df = getPlot()
  f = figure('Visible','off');
  x = -2:0.25:2;
  [X,Y] = meshgrid(x);
  Z = X.*exp(-X.^2-Y.^2);
  contour3(X,Y,Z,30);
  df = webfigure(f);
  close(f);
end
```

- Assume that the function `getPlot` has been deployed in a .NET assembly with a namespace of `MyComponent` and a class of `MyComponentclass`.
- Assume the MATLAB Runtime has been installed. If not, refer to "MATLAB Runtime".
- If you are running on a system with 64-bit architecture, use the information in "Advanced Configuration of a WebFigure" on page 8-9 to work with WebFigures unless you are deploying a website which is 32-bit only and you have a 32-bit MATLAB Runtime installed.

Procedure

To implement WebFigures in .NET for MATLAB Compiler SDK using the Quick Start approach, do the following:

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 In your Visual Studio project, add a reference to *matlabroot*\toolbox \dotnetbuilder\bin*arch*\v2.0\WebFiguresService.dll.

If you are using the MATLAB Runtime, *matlabroot* is the location of the installed MATLAB Runtime.

If you are using MATLAB Compiler SDK, *matlabroot* is the location of the installed MATLAB Compiler SDK.

Note: If you are running on a system with 64-bit architecture, use the information in "Advanced Configuration of a WebFigure" on page 8-9 to work with WebFigures unless you are deploying a website which is 32-bit only and you have a 32-bit MATLAB Runtime installed.

---

5 "Install WebFigureControl Into Microsoft Visual Studio Toolbox" on page 8-2.

6 Drag the WebFigureControl from the toolbox to your web page. After dragging, the web page displays the following default figure.
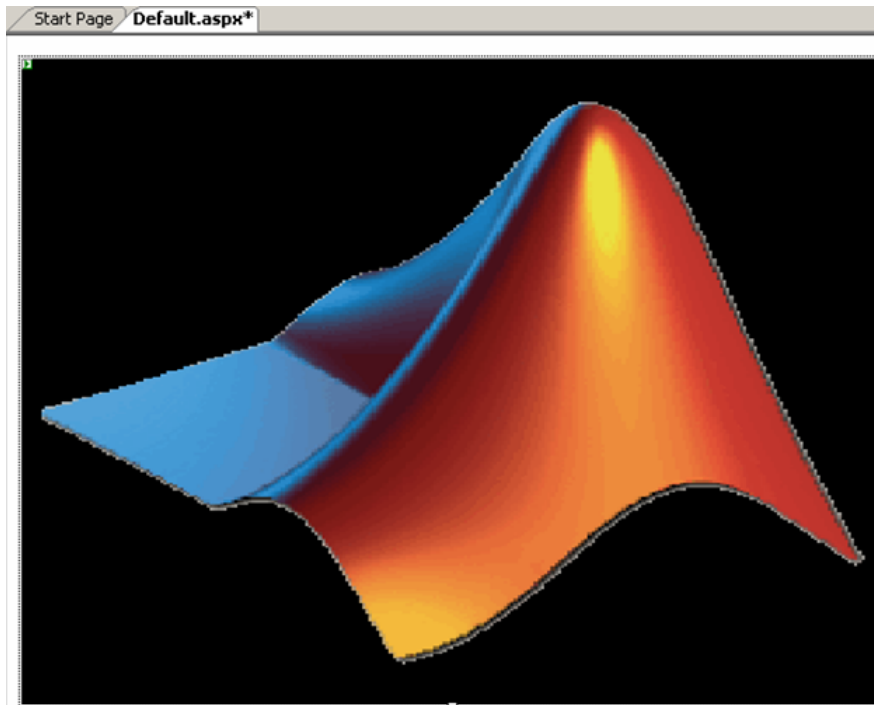
You can resize the control as you would any other .NET web control.

7 Switch to the Design view in Microsoft Visual Studio by selecting **View > Designer**.

8 Test the web page by "playing" it in Microsoft Visual Studio. Select **Debug > Start Debugging**. The page should appear as follows.
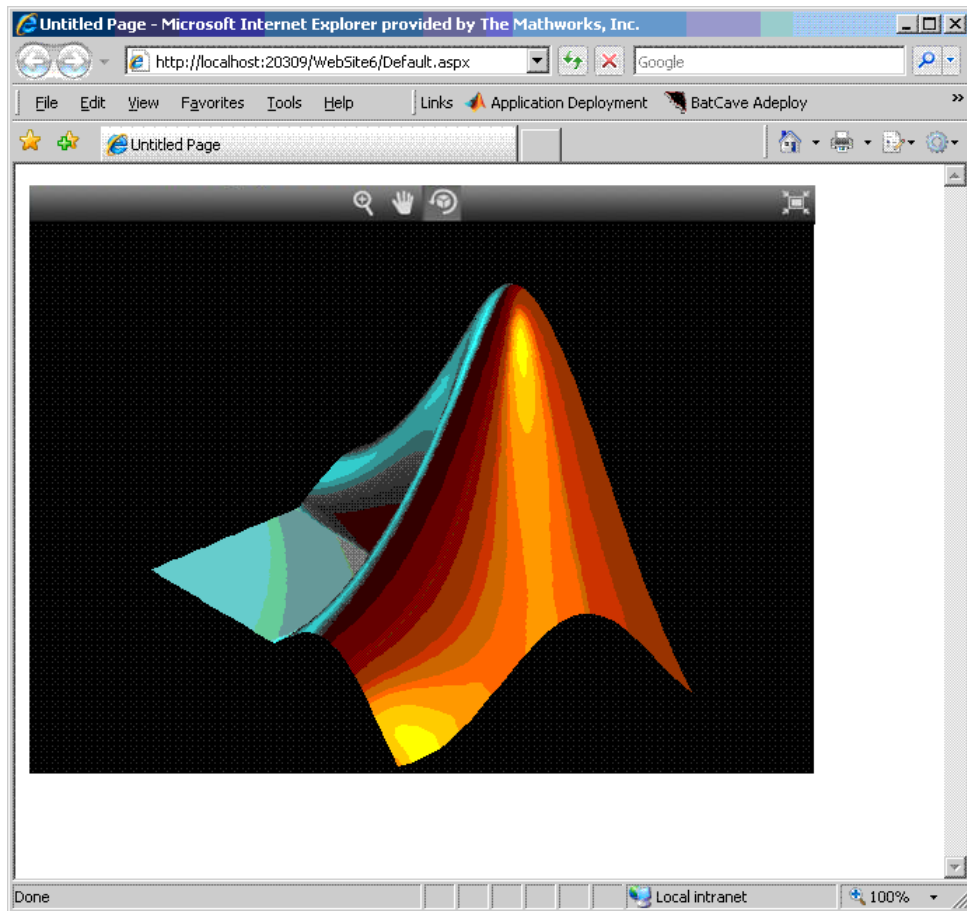
9 Interact with the default figure on the page using your mouse. Click one of the three control icons at the top of the figure to activate the desired control, select the desired region of the figure you want to manipulate, then click and drag as appropriate. For example, to zoom in on the figure, click the magnifying glass icon, then hover over the figure.

10 Close the page as you would any other window, automatically exiting debug or "play" mode.

11 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 Example" on page 8-3). See "Common Integration Tasks and Naming Conventions" on page 2-2 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.getPlot());
}
```

- **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.getPlot())
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 8-21.

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 8-9 for information on creating a custom configuration.

12 Replay the web page in Microsoft Visual Studio to confirm your WebFigure appears as desired.

Advanced Configuration of a WebFigure

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 8-3.

Using the WebFigure Control on the Frontend Servers Outside the Firewall

Presentation Tier
Load Balancer

Company
Boundary

Web servers with WebFigure control
on page and WebFigure service
activated

Shared
Session
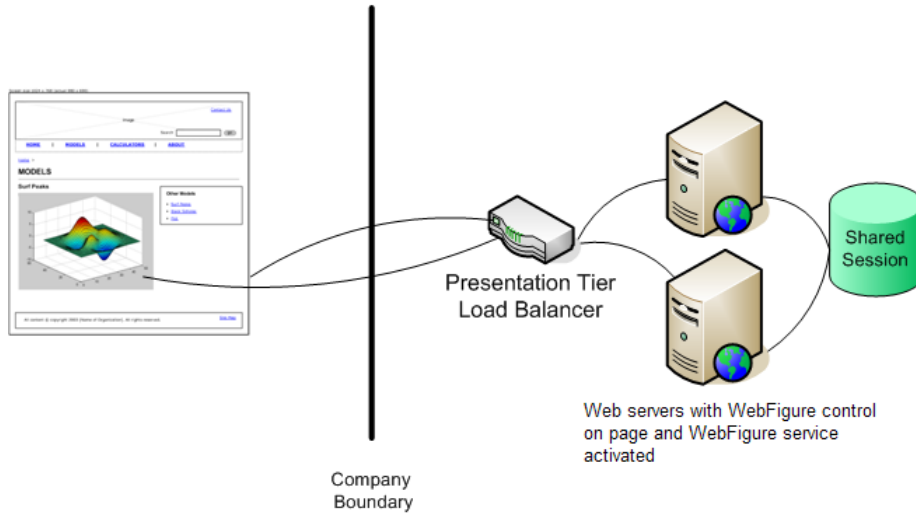
**Using the Remote WebFigure Control to
Keep the WebFigure Service Cluster
Behind the Firewall**
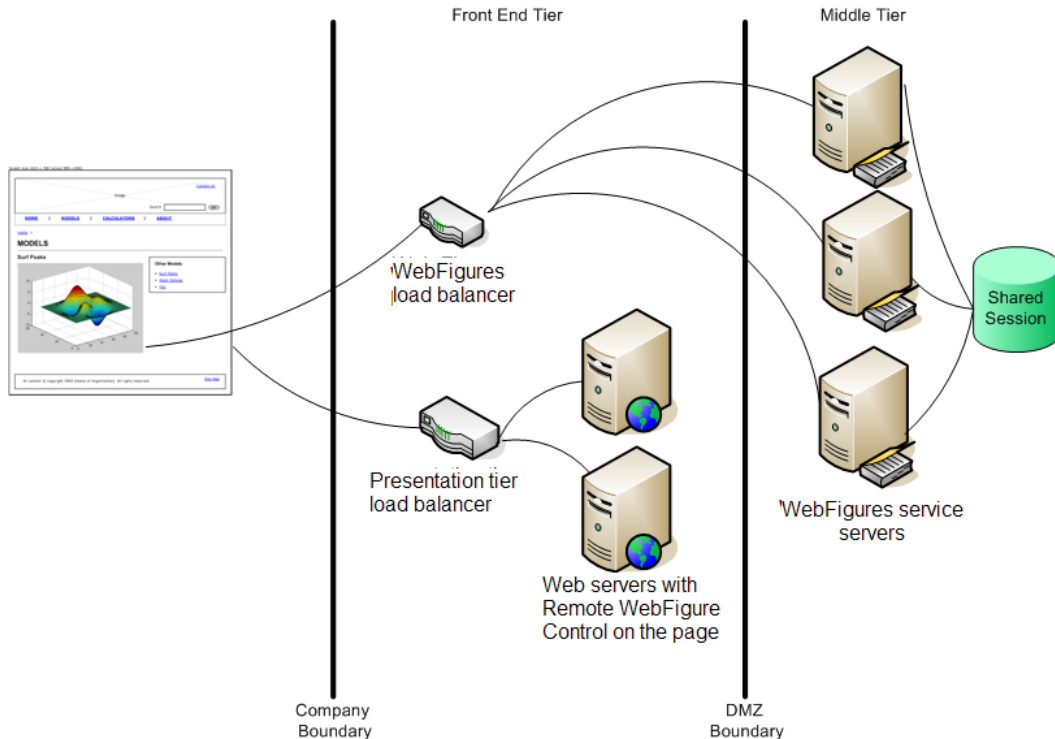
Manually Installing WebFigureService

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 install WebFigureService manually. 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 8-3.

- 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 8-3, you automatically installed `WebFigureService` in the web application file `web.config`.

To install `WebFigureService` manually:

1 Add a reference to `WebFiguresService.dll` from the folder *matlabroot*\toolbox\dotnetbuilder\bin*arch*\v2.0 to the project.

If you are using MATLAB Runtime, *matlabroot* is the location of the installed MATLAB Runtime. If you are using MATLAB, *matlabroot* is the location of the MATLAB installation.

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`:

```
<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 code *must be entered on one continuous line.*

3 To use the 64-bit version of IIS, add the following to `web.config`:

```
<appSettings>
  <add key="PageInspector:ServerCodeMappingSupport" value="Disabled" />
</appSettings>
<system.webServer>
  <validation validateIntegratedModeConfiguration="false" />
  <handlers>
    ...
  </handlers>
</system.webServer>

<compilation debug="true" targetFramework="4.0">
  <assemblies>
```

```
        <add assembly="WebFiguresService,
                       Version=2.14.0.0,
                       Culture=neutral,
                       PublicKeyToken=E1D84A0DA19DB86F"/>
        <remove assembly="Microsoft.VisualStudio.Web.PageInspector.Loader,
                          Version=1.0.0.0,
                          Culture=neutral,
                          PublicKeyToken=b03f5f7f11d50a3a"/>

    </assemblies>
</compilation>
```

Note: The value for the `assembly=` statements in the code *must be entered on one continuous line.*

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.getPlot());
        }
}
```

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.getPlot())
        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.getPlot(4);

                WebFigureControl1.WebFigure =
                        new WebFigure(outputs[0]);

    WebFigureControl2.WebFigure =
                        new WebFigure(outputs[1]);

    WebFigureControl3.WebFigure =
                        new WebFigure(outputs[2]);

      WebFigureControl4.WebFigure =
```

```
                            new WebFigure(outputs[3]);
        }
}
```

Visual Basic

```
Imports MyComponent
Imports MathWorks.MATLAB.NET.WebFigures

Class
        Protected Sub Page_Load(ByVal sender As Object,
                                ByVal e As System.EventArgs)
                                Handles Me.Load
                Dim myDeployedComponent As _
                        New MyComponentclass()

    Dim outputs as MWArray() = _
    myDeployedComponent.getPlot(4)

                WebFigureControl1.WebFigure = _
                        New WebFigure(outputs(0))

                WebFigureControl2.WebFigure = _
                        New WebFigure(outputs(1))

                WebFigureControl3.WebFigure = _
                        New WebFigure(outputs(2))

                WebFigureControl4.WebFigure = _
                        New WebFigure(outputs(3))
        End Sub
End Class
```

Attaching a WebFigure

After you have manually installed `WebFigureService`, the server where it is installed is ready to receive requests for any `WebFigure` information. In the Quick Start, `WebFigureService` uses the session cache built into IIS to retrieve a `WebFigure`, per user, and display it. Since a `WebFigureControl` isn't being used in this case, you need to manually set up the `WebFigureService` and attach the `WebFigure`. Add the code supplied in this section to attach a `WebFigure` of your choosing.

This method of setting up `WebFigureService` and attaching the figure manually is very useful in the following situations:

- You do not want front-end servers to have `WebFigureService` running on them for performance reasons.

- You are displaying a `WebFigure` that does not change based on the current user or session. When multiple users are sharing the same `WebFigure`, which is very common, it is much more efficient to store a single `WebFigure` in the `Application` or `Cache` state, rather than issuing all users their own figure.

There are a number of ways to attach a `WebFigure` to a scope, depending on state (note that these terms follow standard industry definitions and usage):

State	Definition
Session	The method used by `WebFigureControl` by default, which is tied to a specific user session and cannot be shared across sessions. If you use IIS session sharing capabilities, you can use this across servers in a cluster.
Application	Available for any user of your application, per application lifetime. IIS will not propagate this across servers in a cluster, but if each server attaches the data to this cache once, all users can access it very efficiently.
Cache	Similar to `Application`, but with more potential settings. You can assign "time to live" and other settings found in Microsoft documentation.

Note: In this type of configuration, it is typical to have the following code executed once in the `Global.asax` server startup block. For more information on `Global.asax`, see "Using Global Assembly Cache (Global.asax) to Create WebFigures at Server Start-Up" on page 8-21.

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.getPlot());
```

Or

```
Application["ApplicationStateWebFigure"] =
        new WebFigure(myDeployedComponent.getPlot());
```

Or

```
Cache["CacheStateWebFigure"] =
        new WebFigure(myDeployedComponent.getPlot());
```

- **Visual Basic:**

```
Dim myDeployedComponent As _
        New MyComponentclass()

Session("SessionStateWebFigure") = _
        New WebFigure(myDeployedComponent.getPlot())
```

Or

```
Application("ApplicationStateWebFigure") = _
        New WebFigure(myDeployedComponent.getPlot())
```

Or

```
Cache("CacheStateWebFigure") = _
        New WebFigure(myDeployedComponent.getPlot())
```

Setting Up WebFigureControl for Remote Invocation

After you drag a `WebFigureControl` onto a page, as in "Quick Start Implementation of WebFigures" on page 8-3, 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 8-3.

Note: If you are running on a system with 64-bit architecture, use the information in "Advanced Configuration of a WebFigure" on page 8-9 to work with WebFigures unless you are deploying a website 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:
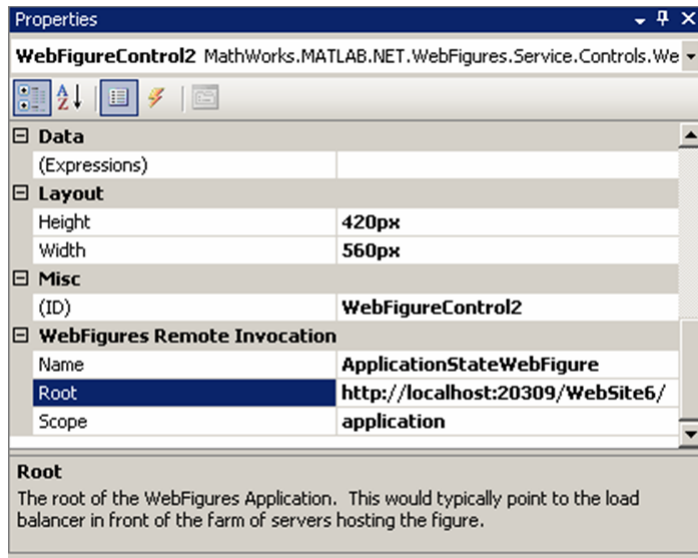
Note: If you don't provide a **root** (usually the location of the load balancer), it is assumed to be the server where the page is executing.

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 8-11 in "Attaching a WebFigure" on page 8-15.

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 8-15.

GetHTMLEmbedString API Parameters

Parameter	If not specified...
ID	Default MATLAB WebFigure (the MATLAB membrane logo).

Parameter	If not specified...
Root	The relative path to the current web page will be used.
WebFigureAttachType	Will search through Session state, then Application state, then Cache state.
Height	Default height will be 420.
Width	Default width will be 560.

Referencing a WebFigure Attached to the Local Server

- **C#:**

```
using MathWorks.MATLAB.NET.WebFigures.Service;

String localEmbedString =
        WebFigureServiceUtility.GetHTMLEmbedString(
                "SessionStateWebFigure",
                WebFigureAttachType.session,
                300,
                300);

Response.Write(localEmbedString);
```

- **Visual Basic:**

```
Imports MathWorks.MATLAB.NET.WebFigures.Service

Dim localEmbedString As String = _
        WebFigureServiceUtility.GetHTMLEmbedString( _
                "SessionStateWebFigure", _
                WebFigureAttachType.session, _
                300, _
                300)

Response.Write(localEmbedString)
```

Referencing a WebFigure Attached to a Remote Server

- **C#:**

```
using MathWorks.MATLAB.NET.WebFigures.Service;

String remoteEmbedString =
```

```
        WebFigureServiceUtility.GetHTMLEmbedString(
                "SessionStateWebFigure",
                "http://localhost:20309/WebSite7/",
                WebFigureAttachType.session,
                300,
                300);
```

```
Response.Write(remoteEmbedString);
```

- **Visual Basic:**

```
Imports MathWorks.MATLAB.NET.WebFigures.Service

Dim localEmbedString As String = _
        WebFigureServiceUtility.GetHTMLEmbedString( _
                "SessionStateWebFigure", _
                "http://localhost:20309/WebSite7/", _
                WebFigureAttachType.session, _
                300, _
                300)
```

```
Response.Write(localEmbedString)
```

Improving Processing Times for JavaScript Using Minification

This application uses JavaScript® to perform most of its AJAX functionality. Because JavaScript runs in the client browser, it must all be streamed to the client computer before it can execute. To improve this process, you use a standard JavaScript minification algorithm to remove comments and white space in the code. This feature is enabled by default. To disable it, create an environment variable called mathworks.webfigures.disableJSMin and set its value to true.

Using Global 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 8-3, 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 8-15 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.getPlot());

    //Or

    Cache["CacheStateWebFigure"] =
        new WebFigure(myDeployedComponent.getPlot());
}
```

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.getPlot())

    'Or
```

```
    Cache("CacheStateWebFigure") = _
        New WebFigure(myDeployedComponent.getPlot())
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 8-17 or directly from the web page as in "Getting an Embeddable String That References a WebFigure Attached to a WebFigureService" on page 8-19.

Upgrade Your WebFigures

If you want to upgrade your version of MATLAB Compiler SDK 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 Compiler SDK.
4 Add the new `WebFigureControl` icon to the toolbox.
5 Drag new WebFigures on to your page.

Troubleshoot WebFigures

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 8-11.
- `WebFigure` is not attached to `WebFigureService`. See "Attaching a WebFigure" on page 8-15.
- Remote root URL is pointing to an invalid server.

Common errors and their diagnosis follow.

Error	Diagnosis
Issue Displaying Image. Please Refresh.	Most often, this message is generated when the session state has expired and the WebFigure has been deleted. Refreshing the session will reestablish the WebFigure in cache and the figure will reappear.
No `WebFigure` Can Be Found with the Name Specified	The `WebFigure` isn't attached correctly. See "Attaching a WebFigure" on page 8-15.
`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.

Error	Diagnosis
`WebFigureService` Not Functioning	The `WebFigureService` `httpHanderFactory` could not be found on the server specified. See "Manually Installing WebFigureService" on page 8-11.
Could not find a part of the path *pathname*	The logging environment variable is set to a folder that does not exist.

WebFigures Logging Levels

There are several logging levels that can be used to diagnose problems with WebFigures.

Logging Level	Uses
`Severe`	Unrecoverable errors and exceptions
`Warning`	Recoverable errors that might occur
`Information`	Informative messages
`Finer`	For monitoring application flow (when different parts of an application are executed)

You can manually set the log level by setting an environment variable called `mathworks.webfigures.logLevel` to one of the above strings.

If you set this environment variable to something other than the above strings or it is not set, it defaults to a level of `Warning` or `Severe` only.

By default, all exceptions are shown within the `WebFigure` control on the web page when `debug` mode is on for the site.

If you want more detailed logging information, or log information when `debug` is not on, set an environment variable called `mathworks.webfigures.logLocation` to the location where the log file is written. The log file is named *yourwebappname*`WFSLog.txt`.

Create and Modify a MATLAB Figure

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', [O, O, width, height]);
view([rotation, elevation]);
returnByteArray = figToImStream(`figHandle', h, ...
                                `imageFormat', imageFormat, ...
                                `outputType', `uint8');
close(h);

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', [O, O, width, height]);
view([rotation, elevation]);
returnByteArray = figToImStream(`figHandle', h, ...
                                `imageFormat', imageFormat, ...
                                `outputType', `int8');
close(h);
```

Work with Images

Getting Encoded Image Bytes from an Image in a Component

```
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

```
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, O,
    myByteArray.Length);
    ms.Write(myByteArray, O, 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.

```
WebFigure figure =
    new WebFigure(deployment.getWebFigure());
WebFigureRenderer renderer =
    new WebFigureRenderer();

//Creates a parameter object that can be changed
// to represent a specific WebFigure and its orientation.
//If you dont set any values it uses the defaults for that
// figure (what they were when the figure was created in M).
WebFigureRenderParameters param =
    new WebFigureRenderParameters(figure);

param.Rotation = 30;
param.Elevation = 30;
param.Width = 500;
param.Height = 500;

//If you need a byte array that can be streamed out
// of a web page you can use this:
byte[] outputImageAsBytes =
    renderer.RenderToEncodedBytes(param);
```

```
//If you need a .NET Image (can't be used on the web)
// you can use this code:
Image outputImageAsImage =
    renderer.RenderToImage(param);
```

9

Windows Communications Foundation Based Components

What Is Windows Communications Foundation?

Windows Communication Foundation (WCF) is an application programming interface in the .NET Framework for building service-oriented applications. Servers implement 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. A WCF client connects to a service via an *endpoint*. Each service exposes itself 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?

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."

Create Windows Communications Foundation Based Components

In this section...
"Before Running the Example" on page 9-3
"Deploying a WCF-Based Component" on page 9-3

Before Running the 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 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 9-4
2. "Develop Your WCF Interface" on page 9-4
3. "Build Your Component and Generate Your Type-Safe API" on page 9-5
4. "Develop Server Program Using the WCF Interface" on page 9-7
5. "Compile the Server Program" on page 9-10
6. "Run the Server Program" on page 9-10
7. "Generate Proxy Code for Clients" on page 9-11
8. "Compile the Client Program" on page 9-12
9. "Run the Client Program" on page 9-14

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 "Implement 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 "Implement a Type-Safe Interface" on page 7-10.

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

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 app or the deployment command line tools to generate the type-safe API.

Using the Library Compiler

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

1   Create your 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 Library Compiler app 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 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 Compiler SDK assembly 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 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 2-22 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="MexHtppBinding"/>
      <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:
`IAddOne.dll`	The .NET native type interface `IAddOne`
`MechanismIAddOne.dll`	The generated type-safe API
`AddOneCompNative.dll`	The generated assembly

> **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 9-12 for information about modifying port configurations.

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

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
```

```csharp
        Console.WriteLine("Conntecting to
                AddMaster Service through
                Http connection...");
        AddOneClient AddMaster =
                new AddOneClient("HttpBinding");
        Console.WriteLine("Conntected 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:

```
Conntecting to AddMaster Service through Http connection...
Conntected 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....
```

10

.NET Remoting

What Is .NET Remoting?

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.

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?

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.

.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.
- It is important to determine if you derive more benefit and cost savings by using the `MWArray` API or the native .NET API. Evaluate if .NET Remoting is appropriate for your deployable component by reading "Select How to Access an Assembly" on page 10-5.

Select How to Access an Assembly

There are two data conversion API's that are available to marshal and format data across the managed (.NET) and 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 Compiler SDK. 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 "Access a Remotable .NET Assembly Using MWArray" on page 10-10.

The Native API was designed especially, though not exclusively, to support .NET remoting. It allows you to pass arguments and return values using standard .NET types. This feature is especially useful for clients that access a remoteable component using the native interface API, as it does not require the client machine to have the MATLAB 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 "Access a Remotable .NET Assembly Using the Native .NET API: Magic Square" on page 10-16.

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 "Access a Remotable .NET Assembly Using the Native .NET API: Cell and Struct" on page 10-23).		

Using Native .NET Structure and Cell Arrays

The MATLAB Compiler SDK 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 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 Compiler SDK provides a special class hierarchy for struct and cell array representation designed to easily interface with the native .NET API.

See "Access a Remotable .NET Assembly Using the Native .NET API: Cell and Struct" on page 10-23 for details.

Create a Remotable .NET Assembly

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\VS*Version*\NET\MagicRemoteExample\MWArr

     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\VS*Version*\NET\MagicRemoteExample\Nativ

     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)

    y = magic(x);
```

3 Click the **Library Compiler** app in the apps gallery.

4 In the Additional Runtime Settings area, select **Enable .NET Remoting**.
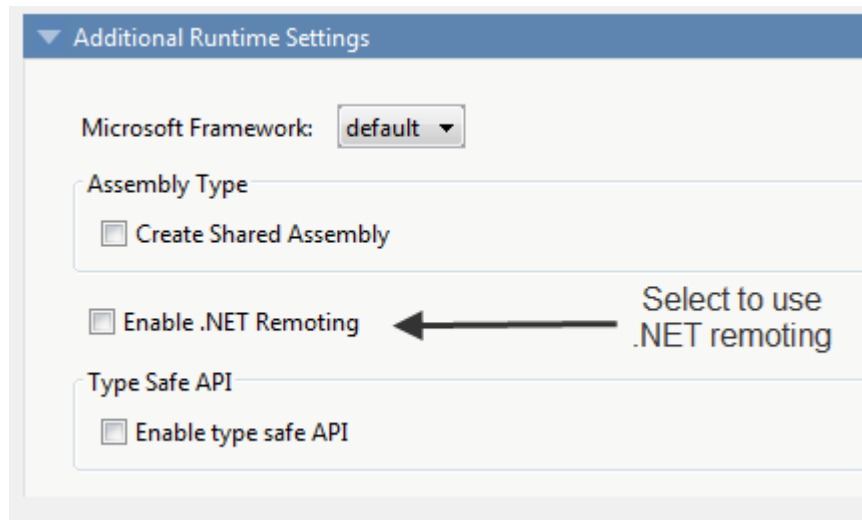
5 Build the .NET component. See the instructions in "Create a .NET Assembly" for more details.

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.

Access a Remotable .NET Assembly Using MWArray

Why Use 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 How to Access an Assembly" on page 10-5.

Some reasons you might use the MWArray API instead of the native .NET API are:

- You are working with data structure arrays, which the native .NET API does not support.
- You or your users work extensively with many MATLAB data types.
- You or your users are familiar and comfortable using the MWArray API.

For information on accessing your component using the native .NET API, see "Access a Remotable .NET Assembly Using the Native .NET API: Magic Square" on page 10-16.

Coding and Building the Hosting Server Application and Configuration File

The server application hosts the remote component built in "Create a Remotable .NET Assembly" on page 10-7. You can also perform these steps using the native .NET API as discussed in "Access a Remotable .NET Assembly Using the Native .NET API: Magic Square" on page 10-16.

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 10-10.

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:

```csharp
using System;
using System.Configuration;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

using System.Collections;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Remoting.Channels.Tcp;

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

using IMagicSquareComp;

namespace MagicSquareClient
{
  class MagicSquareClient
    {
      static void Main(string[] args)
        {
          try
            {
              RemotingConfiguration.Configure
                                    (@"MagicSquareClient.exe.config");

              String urlServer=
                  ConfigurationSettings.AppSettings["MagicSquareServer"];

              IMagicSquareClass magicSquareComp=
                (IMagicSquareClass)Activator.GetObject
                                  (typeof(IMagicSquareClass),
                                   urlServer);

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

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

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

```
            }

          catch (Exception exception)
            {
              Console.WriteLine(exception.Message);
            }

          Console.ReadLine();
        }
    }
}
```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remoteable component.
- The client instantiates the remoteable object using the static `Activator.GetObject` method
- From this point, the remoting client calls methods on the remoteable component exactly as it would call a local component method.

MagicSquareClient Configuration File

The configuration file for the magic square client is in the file `MagicSquareClient \MagicSquareClient.exe.config`. The configuration file, written in XML, is shown here:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MagicSquareServer"
     value="tcp://localhost:1234/MagicSquareClass.remote"/>
  </appSettings>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel name="MagicSquareChannel" ref="tcp" port="0">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

This code specifies:

- The name of the remote component server and the remote component URI (uniform resource identifier)
- The remoting protocol (`TCP/IP`) and port number
- The message formatter (`binary`) and the permissions for the communication channel (`full` trust)

Starting the Server Application

Starting the server by doing the following:

1 Open a DOS or UNIX command window and `cd` to `MagicSquareServer\bin\x86\v2.0\Debug`.

2 Run `MagicSquareServer.exe`. You will see the message:

```
Magic Square Server started...
```

Starting the Client Application

Start the client by doing the following:

1 Open a DOS or UNIX command window and `cd` to `MagicSquareClient\bin\x86\v2.0\Debug`.

2 Run `MagicSquareClient.exe`. After the MATLAB Runtime initializes, you should see the following output:

```
Magic square of order 4

162313
511108
97612
414151
```

Access a Remotable .NET Assembly 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 How to Access an Assembly" on page 10-5.

Some reasons you might use the native .NET API instead of the MWArray API are:

- You want to pass arguments and return values using standard .NET types, and you or your users don't work extensively with data types specific to MATLAB.
- You want to access your component from a client machine without an installed version of MATLAB.

For information on accessing your component using the MWArray API, see "Access a Remotable .NET Assembly Using MWArray" on page 10-10.

Coding and Building the Hosting Server Application and Configuration File

The server application will host the remote component you built in "Create a Remotable .NET Assembly" on page 10-7.

The client application, running in a separate process, will access the remote component hosted by the server application. Build the server with the Microsoft Visual Studio project file MagicSquareServer\MagicSquareServer.csproj:

1 Change the reference for the generated component assembly to MagicSquareComp \for_redistribution_files_only\MagicSquareCompNative.dll.

2 Select the appropriate build platform.

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:

```csharp
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
<?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 10-16. Build the remote client using the Microsoft Visual Studio project file `MagicSquareClient \MagicSquareClient.csproj`. To create the remote client using Microsoft Visual Studio:

1 Change the reference for the generated component assembly to `MagicSquareComp \for_redistribution_files_only\MagicSquareCompNative.dll`.

2 Change the reference for the generated interface assembly to `MagicSquareComp \for_redistribution_files_only\IMagicSquareCompNative.dll`.

3 Select the appropriate build platform.

4 Select **Debug** or **Release** mode.

5 Build the `MagicSquareClient` project.

6 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:

```csharp
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
<?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 cdto 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
```

Access a Remotable .NET Assembly 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\VS*Version*\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 10-7 or "Building a Remotable Component Using the mcc Command" on page 10-8.

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_name*Native.dll.

2 Select the appropriate build platform.

3 Select **Debug** or **Release** mode.

4 Build the NativeStructCellServer project.

5 Supply the configuration file for the NativeStructCellServer. The C# code for the server is in the file NativeStructCellServer.cs:

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

namespace NativeStructCellServer
{
  class NativeStructCellServer
    {
      static void Main(string[] args)
        {
          RemotingConfiguration.Configure(
                         @"NativeStructCellServer.exe.config");

          Console.WriteLine("NativeStructCell Server started...");

          Console.ReadLine();
        }
    }
}
```

This code reads the associated configuration file to determine:

- Name of the component to host

- Remoting protocol and message formatting to use

- Lease time for the remote component

In addition, the code also signals that the server is active and waits for a carriage return before terminating.

Coding and Building the Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application built in "The Native .NET Cell and Struct Example" on page 10-23. Build the remote client using the Microsoft Visual Studio project

file `NativeStructCellClient\NativeStructCellClient.csproj`. To create the remote client using Microsoft Visual Studio:

1 Change the references for the generated component assembly to *component_name*`\for_redistribution_files_only` `\`*component_name*`Native.dll`.

2 Change the references for the generated interface assembly to *component_name*`\for_redistribution_files_only` `\I`*component_name*`Native.dll`.

3 Select the appropriate build platform.

4 Select **Debug** or **Release** mode.

5 Build the `NativeStructCellClient` project.

6 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

Failure to Find MATLAB Runtime Files

If your application generates a diagnostic message indicating that a module cannot be found, it could be that the MATLAB Runtime is not properly located on your path. You fix this problem by ensuring that the MATLAB Runtime files are on your application path.

On a system with MATLAB installed, *matlabroot*\runtime*arch* must be on your system path ahead of any other MATLAB installations.

- *matlabroot* is your root MATLAB folder.
- *arch* is the architecture of your computer such as win64 for a 64-bit Windows computer.

On a system with MATLAB Runtime installed, *mcr_root**ver*\runtime*arch* is on your system path.

- *mcr_root* is your root MATLAB Runtime folder. *ver* is the version number of the MATLAB Runtime.
- *arch* is the architecture of your computer such as win64 for a 64-bit Windows computer.

Failure to Find MATLAB Classes

If your application generates a diagnostic message indicating that
Mathworks.MATLAB.NET.*name* cannot be found, it could be that you need to reference
the MWArray.dll assembly in your Visual Studio project.

The MWArray.dll assembly is located at *matlabroot*\toolbox\dotnetbuilder\bin
arch\v2.0.

- *matlabroot* is your root MATLAB or MATLAB Runtime folder.
- *arch* is the architecture of your computer such as win64 for a 64-bit Windows
  computer.

Diagnostic Messages

The following table shows diagnostic messages you might encounter, probable causes for the message, and suggested solutions.

See the following table for information about some diagnostic messages.

Diagnostic Messages and Suggested Solutions

Message	Probable Cause	Suggested Solution
`LoadLibrary("`*`component_nam`* `failed - The specified module could not be found.`	You may get this error message while registering the project DLL from the DOS prompt. This can occur if the MATLAB Runtime is not on the system path.	See "Failure to Find MATLAB Runtime Files" on page 11-2.
`Error in` *`component_name.class_name.`* `Error getting data conversion flags.`	This is often caused by `mwcomutil.dll` not being registered.	**1** Open a DOS window. **2** Change folders to *`matlabroot`*`\runtime` *`\architecture`*. **3** Run the following command:`mwregsvrmwcomutil`.dll (*`matlabroot`* is your root MATLAB folder.)
`Error in VBAProject: ActiveX component can't create object.`	• Project DLL is not registered. • An incompatible MATLAB DLL exists somewhere on the system path.	If the DLL is not registered, **1** Open a DOS window. **2** Change folders to *`projectdir`*`\distrib`. **3** Run the following command:`mwregsvr`*`projectdll`*`.dll` (*`projectdir`* represents the location of your project files).

Message	Probable Cause	Suggested Solution
`object ref not set to instance of an object`	This occurs when an object that has not been instantiated is called	Instantiate the object.
`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 MATLAB Runtime Files" on page 11-2.
`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.`	This warning occurs when ASP.NET code tries to bring up a dialog box. If occurs because `getframe()` makes the figure window visible before performing the capture and thus fails when running in IIS. `msgbox()` calls in MATLAB code cause the warning to appear also.	Work around this problem by doing the following: **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\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name
dmakeerror_error2,line at 14.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name
dmakeerror_error1,line at 11.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name dmakeerror,line at 4.
    at
file H:\compiler\g388611\cathy\MagicDemoCSharpApp\bin\Debug\
CalldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\calldmakeerror.m,name
calldmakeerror,line at 2.

... .Application Stack Trace ...
   at MathWorks.MATLAB.NET.Utility.MWMCR.EvaluateFunction
(String functionName, Int32 numArgsOut, Int
32 numArgsIn, MWArray[] argsIn)
   at MathWorks.MATLAB.NET.Utility.MWMCR.EvaluateFunction
(Int32 numArgsOut, String functionName, MWA
rray[] argsIn)
   at CalldmakeerrComp.Calldmakeerr.calldmakeerror() in
h:\compiler\g388611\cathy\MagicDemoComp\src\
Calldmakeerr.cs:line 140
   at MathWorks.Demo.MagicSquareApp.MagicDemoApp.Main(String[]
 args) in H:\compiler\g388611\cathy\Ma
gicDemoCSharpApp\MagicDemoApp.cs:line 52
```

Reference Information

Rules for Data Conversion Between .NET and MATLAB

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 makeDouble constructor argument is set to false. The default is true, 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	Conversion to the equivalent unsigned type is not supported
uint32	System.Int32	MWNumericArray	Conversion to the equivalent unsigned type is not supported
uint16	System.Int16	MWNumericArray	Conversion to the equivalent unsigned type is 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 Compiler SDK 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 • 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,..,Pn]
Scalar .NET struct	MATLAB struct constructed from `public` instance fields of the .NET struct

.NET Type	Converts to MATLAB Type
.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
`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.
	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 is an int[,], and 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

To Convert This MATLAB Type:	To this:	Follow these rules:
		`Hashtable`. If the nested `struct` is not a scalar, then an `ArgumentException` is thrown. The dictionary key must be of type `String`.

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 Compiler SDK 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 Compiler SDK 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 12-10
"Returning Data from MATLAB to Managed Code" on page 12-10
"Example of MWNumericArray in a .NET Application" on page 12-11
"Interfaces Generated by the MATLAB Compiler SDK .NET Target" on page 12-11

Overview

The data conversion classes are

- `MWArray`
- `MWIndexArray`
- `MWCellArray`
- `MWCharacterArray`
- `MWLogicalArray`
- `MWNumericArray`
- `MWStructArray`

`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 Compiler SDK .NET Target

For each MATLAB function that you specify as part of a .NET assembly, the MATLAB Compiler SDK product generates an API based on the MATLAB function signature, as follows:

- A *single output* signature that assumes that only a single output is required and returns the result in a single `MWArray` rather than an array of `MWArray`.
- 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 MATLAB Compiler SDK product 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`.

For example, the following table shows a generic function `foo` along with the single output API that the compiler generates for its several forms.

Generic MATLAB function	`function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)`
API if there are no input arguments	`public MWArray foo()`
API if there are one or more input arguments	`public MWArray foo(MWArray In1, MWArray In2...MWArray inN)`
API if there are optional input arguments	`public MWArray foo(MWArray In1, MWArray In2,     ..., MWArray inN` `params MWArray[] varargin)`

In the example, the input arguments *In1, In2*, and *inN* are of type `MWArray`.

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` rather than a single array object.

The standard API for a generic function with none, one, more than one, or a variable number of arguments, is shown in the following table.

Generic MATLAB function	`function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)`
API if there are no input arguments	`public MWArray[] foo(int numArgsOut)`
API if there is one input argument	`public MWArray [] foo(int numArgsOut, MWArray In1)`
API if there are two to *N* input arguments	`public MWArray[] foo(int numArgsOut, MWArray In1,MWArray In2, \... MWArray InN)`
API if there are optional arguments, represented by the `varargin` argument	`public MWArray[] foo(int numArgsOut,MWArray in1,MWArray in2, MWArray InN, params MWArray[] varargin)`

Details about the arguments for these samples of standard signatures are shown in the following table.

Argument	Description	Details
numArgsOut	Number of outputs	An integer indicating the number of outputs you want the method to return. The value of *numArgsOut* must be less than or equal to the MATLAB function `nargout`. The `numArgsOut` argument must always be the first argument in the list.
In1, In2, ...InN	Required input arguments	All arguments that follow *numArgsOut* in the argument list are inputs to the method being called.

Argument	Description	Details
		Specify all required inputs first. Each required input must be of type `MWArray` or one of its derived types.
varargin	Optional inputs	You can also specify optional inputs if your MATLAB code uses the `varargin` input: list the optional inputs, or put them in an `MWArray[]` argument, placing the array last in the argument list.
`Out1, Out2, ...OutN`	Output arguments	With the standard calling interface, all output arguments are returned as an array of `MWArray`.

feval API

In addition to the methods in the single API and the standard API, in most cases, the MATLAB Compiler SDK product produces an additional overloaded method. If the original MATLAB code contains no output arguments, then the compiler will not generate the `feval` method interface.

For a function with the following structure:

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

The compiler generates the following API, known as the *feval interface*,

```
public void foo
            (int numArgsOut,
             ref MWArray [] ArgsOut,
             MWArray[] ArgsIn)
```

where the arguments are as follows:

numArgsOut	Number of outputs	Same as standard interface.
		An integer indicating the number of outputs you want to return.
		This number generally matches the number of output arguments that follow.

		The `varargout` array counts as just one argument, if present.
`ref MWArray [] ArgsOut`	Output arguments	Following `numArgsOut` are all the outputs of the original MATLAB code, each listed in the same order as they appear on the left side of the original MATLAB code. A `ref` attribute prefaces all output arguments indicating that these arrays are passed by reference.
`MWArray[] ArgsIn`	Input arguments	`MWArray` types or a supported .NET primitive type. When you pass an instance of an `MWArray` type, the underlying MATLAB array is passed directly to the called function. Native types are first converted to `MWArray` types.

Function Reference

enableTSUtilsfromNetworkDrive
ntswrap

enableTSUtilsfromNetworkDrive

Sets the trust setting to load .NET assemblies from network drive

Syntax

```
enableTSUtilsfromNetworkDrive
```

Description

`enableTSUtilsfromNetworkDrive` sets the trust setting so that the MATLAB Compiler SDK module can load .NET assemblies from remote drives.

Note: This is only required when using .NET 2.0 or 3.5.

Tips

- Administrator privileges are required to run this command.

Examples

To enable use of MATLAB Compiler SDK on a system, enter the following on the MATLAB command line after logging in with Administrator privileges:

```
enableTSUtilsfromNetworkDrive
```

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 "Generate the Type-Safe API with an Assembly" on page 7-8 with a
MATLAB Compiler SDK 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_NET_assembly*.dll

    Path to folder containing .NET assembly that defines component referenced by -c
switch

-c *component_class_name*

    Namespace-qualified name of 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 specfed 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` v*x.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 Compiler SDK class `Mechanism` in the namespace `AddOneCompNative`. By default, `ntswrap` compiles the source code into an assembly `MechanismIAddOne.dll`.

Deploying .NET Components With the F# Programming Language

Magic Square 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".

You must be running Microsoft Visual Studio 2010 or higher to use this example.

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".

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 "MATLAB Runtime" on page 6-2 for information about deploying your component to end users.